The Graph-Oriented Programming Paradigm

Olivier Rey

olivier.rey@mines-paris.org

October 26 2016
Preliminary Version

Abstract

Graph-oriented programming is a new programming paradigm that defines a graph-oriented way to build enterprise software, using directed attributed graph databases on the backend side.

This programming paradigm is cumulating the benefits of several other programming paradigms: object-orientation, functional programming, design by contract, rule-based programming. However, it is consistent in itself and does not depend on any other programming paradigms. It is also *simpler* and *more intuitive* than the other programming paradigms. Moreover, it shows astonishing properties in terms of software evolution and extensibility.

This programming paradigm enables to develop long lasting business applications that *do not generate any technical debt*.

It provides a radically different answer to the maintenance and evolutions phases compared to other programming paradigms. Its use is particularly adapted for applications that must manage high complexity, evolving regulations and/or high numbers of business rules.

With graph-oriented programming, software can evolve structurally without having to redesign it, to migrate the data or to perform non regression testing on it.

In this article, we will explain how graph-oriented programming paradigm can change the way the software industry thinks about software and we will uncover some of the many advantages of the graph-oriented programming paradigm.

1 Introduction

1.1 A New Programming Paradigm

The exercise of introducing a new programming paradigm is difficult. Indeed, in order to be able to explain the new paradigm, we have to compare it to other programming paradigms.

We will suppose, in this article, that the reader has notions of object-oriented programming[1] [19], functional programming [5, 12, 18, 2, 26], design by contract [19], rule-based programming[15] and semantic design [33].

We also suppose that the reader has a basic knowledge of attributed graph databases [20] and a good understanding of basic relational databases mechanisms.

[1] Abbreviated OOP in the article.

1.2 Common Use Cases for Attributed Directed Graph Databases

At the time this article is being written, attributed directed graph databases are mostly used:

- In Internet social applications,
- For Big Data purposes,
- For recommendation algorithms in retail or assimilated businesses,
- In fraud detection based on pattern matching,
- In some other more restricted areas such as reference data management, identity management or network modeling.

This article will not speak about those common graph database use cases, and we will direct the reader to the massive documentation available on the Internet on those topics.

1.3 What is Enterprise Software?

In this article, we will only focus on *enterprise software*[2]. This does not mean that the graph-oriented programming paradigm may not be applicable to other fields of software engineering, but we want to restrict our study to this particular domain.

For us, the enterprise applications have the following characteristics:

1. They manipulate business concepts and they implement part of or complete business processes;
2. They embed business rules, most of the time inside the code;
3. They are submitted to various administrative constraints, standards, laws or regulations;
4. They are bound to take into account regulatory changes, that may be frequent;
5. They are used by several people inside the same company, group or set of companies;

[2] Also called *enterprise business applications*, *enterprise applications* or just *business applications*.

6. They use at least one database (currently mostly relational databases);

7. They are generally proposing *interactive* services through GUI[3] (which is implemented by a transactional mode on the server side) and quite often *batch* services.

We will not consider in this article the production environment for those applications (in house, hosted, Cloud-based, etc.).

In this article, ERP-like[4] software is also considered as being enterprise software.

2 Maintenance and Evolutions in Enterprise Software

In this part, we will come back on well-known current issues faced by the professional software engineering during the maintenance and evolution phases. This will help us understand why graph-oriented programming is so promising for the coming years in the field of enterprise software.

2.1 Coupling in Enterprise Applications

Enterprise applications generate a lot of *coupling*.

As we will see, those couplings are fundamentally attached to the way the software industry is building software. We will show how our current technologies cannot but generate coupling from the very beginning of software construction.

As we saw, an enterprise software is dealing with business concepts and relationships between those business concepts, and has to persist both the concepts and the relationships inside a database.

In this article, we will just analyze the coupling generated by the current dominant programming model: the object-oriented programming model and its persistence through a relational database[5].

2.1.1 The Case of a Simple `0..1` Aggregation

In order to represent the issue of coupling, we will propose several diagrams in which we see the three levels of thinking that we generally use to build enterprise software:

1. The *conceptual level*, also called the analysis level [9];

2. The *code level*, also called the software level represented in our samples by UML design diagrams;

3. The *database level*, also called storage level.

In Fig. 1, we can study the coupling that we generate to represent a simple `0..1` aggregation: the class `A` "knows" the class `B`. By "knowing", we mean that, even if members are not accessed directly, the method `methodA1` will know how to navigate from the instance of `A` to the instance of `B` through the `A` member `myB`.

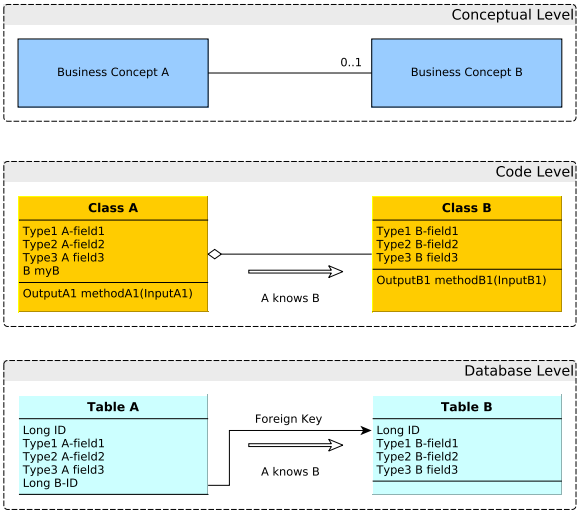

Figure 1: Case of a simple aggregation `0..1`

In the context of our sample, we will suppose, the `methodA1` calls the `methodB1` on the `myB` instance (we are taking a very simple case of *encapsulation* [11]). For sure, in that case, `methodA1` will know the signature of `methodB1` and know how to call it.

We can note that this simple case of encapsulation implies that `A` "owns" two types of knowledge on `B`: the knowledge of navigating from `A` to `B` and the knowledge of `methodB1` interface.

Inside the database, this coupling will be the same as in the code. Where `myB` is something like a pointer to a `B` instance, the relational table representing `A` will contain a column in which we will find the unique identifiers (ID) of `B`. This is often called *foreign key*[6].

This case is really very simple and we see that we have generated, by construction, a coupling at the code and the database level between respectively `A` and `B` classes and `A` and `B` tables. In that case however, we can note that `B` is still autonomous: it is referenced by `A` in the code and inside the database but it does not "know" `A`. We have a single direction coupling: $\texttt{A} \rightarrow \texttt{B}$.

2.1.2 The Case of the 1..n Aggregation

The case of the $1..n$ aggregation is more problematic as we can see in Fig. 2.

In terms of code, we still have a double knowledge at the `methodA1` level: the way of navigating from the instance of `A` to the instances of `B`, and the knowledge of the `methodB1` signature (same hypothesis than in section 2.1.1).

[3] Graphical User Interface.

[4] Enterprise Resource Planning.

[5] Analysis of older enterprise software (such as Cobol applications running managing data in indexed sequential databases) is also very interesting but it is not in the scope of this article.

[6] Indeed, the real foreign key in a relational database brings some enforcement rules with it, such as the non capability of referencing an ID of B if this ID does not correspond to a real instance of B. We will just name *foreign key* the global concept without willing to attach too many database features to it.

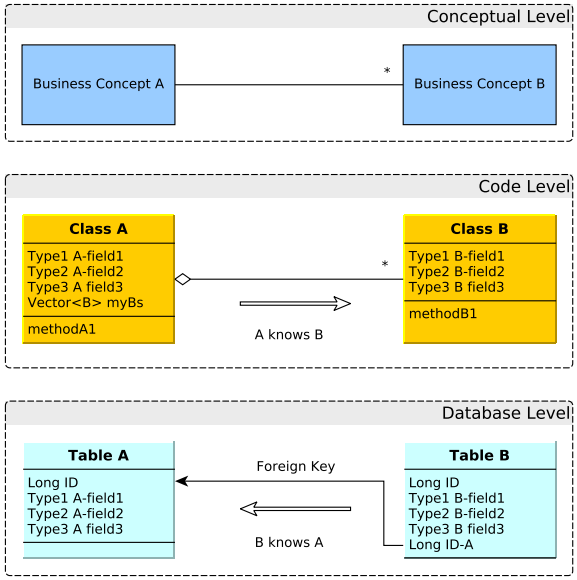

Figure 2: Case of the $0..n$ aggregation

But the database implementation is far worse: the table representing `B` contains a column referencing `A` IDs, which is indeed a reverse dependency. Through the foreign key, the table representing `B` "points" to records in `A`.

This is a very annoying coupling because it is *both ways*: `A` ↔ `B`. `A` knows `B` inside the code and the `B` table knows the `A` table inside the database.

Actually, it seems puzzling that the software industry accepted for so long such a structural problem because, conceptually, `B` remains "pointed to" by `A` and not the reverse.

We can say that the technologies that we use, object-oriented programming and relational databases in that case, structurally generate useless and problematic couplings inside the code and the database.

2.1.3 Large Scale Couplings

Most enterprise software have to manipulate dozens to hundreds of business concepts. When each conceptual relationship is generating a coupling between entities both at the code and at the database level, from the very beginning of the application design, the software engineers have to cope with lots of coupling, most of them being the consequence of technology and not the consequence of the conceptual requirements.

Those couplings will be quite hard to manage during the maintenance evolution phases and it seems that, for a long time, the industry has learned to live with it [23, 25, 24].

2.2 The Necessary Scope Changes

Application scope changes with time to accommodate new requirements and new business practices.

We will define scope as follows: the scope of a software is the intersection between the *business semantic space* and the points of view of some users using the system to perform the business processes the software is supposed to automate.

As we wee in Fig. 3, every *unexpected evolution* can have a massive impact on the application[7]. The Views 1 and 2 are showing a scope evolution that does not touch the core of the system, while Views 3 and 4 show a core system evolution.

If we consider the Fig. 3 View 1, we can see a new requirement coming inside an application scope. This new requirement is not semantically located at the core of the application. It creates the following impacts (View 2):

1. Addition of new software (white circle in Fig. 3);
2. Modifications in existing software (orange circles in Fig. 3);
3. Glue between existing software and new software (light blue circles in Fig. 3);
4. Modifications in new software due to the glue (green circles in Fig. 3).

View 4 shows the impact of a core modification. Generally, this kind of scope evolution is so risky that every other solution will be considered as better (especially if the software engineers can convince the customer the new requirement can be done in another way than through software modification).

Generally, in the maintenance and evolution phases, software engineers feel that they are not free to make the application evolve as it should. If most of them think the previous maintenance teams did a bad job, the experience makes many of them to realize that there is no legacy application that is "well designed" and that this statement may imply a deeper problem.

2.3 The Multiple Attempts of the Industry to Reduce Coupling And Accommodate Scope Changes

For decades, the enterprise software industry have tried a lot of empiric methods to reduce coupling and accommodate scope changes:

1. The use of software design patterns [11];
2. The development of software architecture concerns [4, 32, 30, 1, 8];
3. The development of refactoring methods [3, 10];
4. The use of ORM[8] systems and generated code and/or queries.

Instead of being real solutions, those attempts can be seen mainly as *work-arounds* solving partially the maintenance and evolution problems.

We will very briefly highlight the issues with those various work-arounds. Every topic would deserve much more explanation but we will try to stick to what we consider as being the main points.

[7] For certain categories of applications, regulatory changes are the main source of unexpected evolutions.

[8] Object-Relational Mapping.

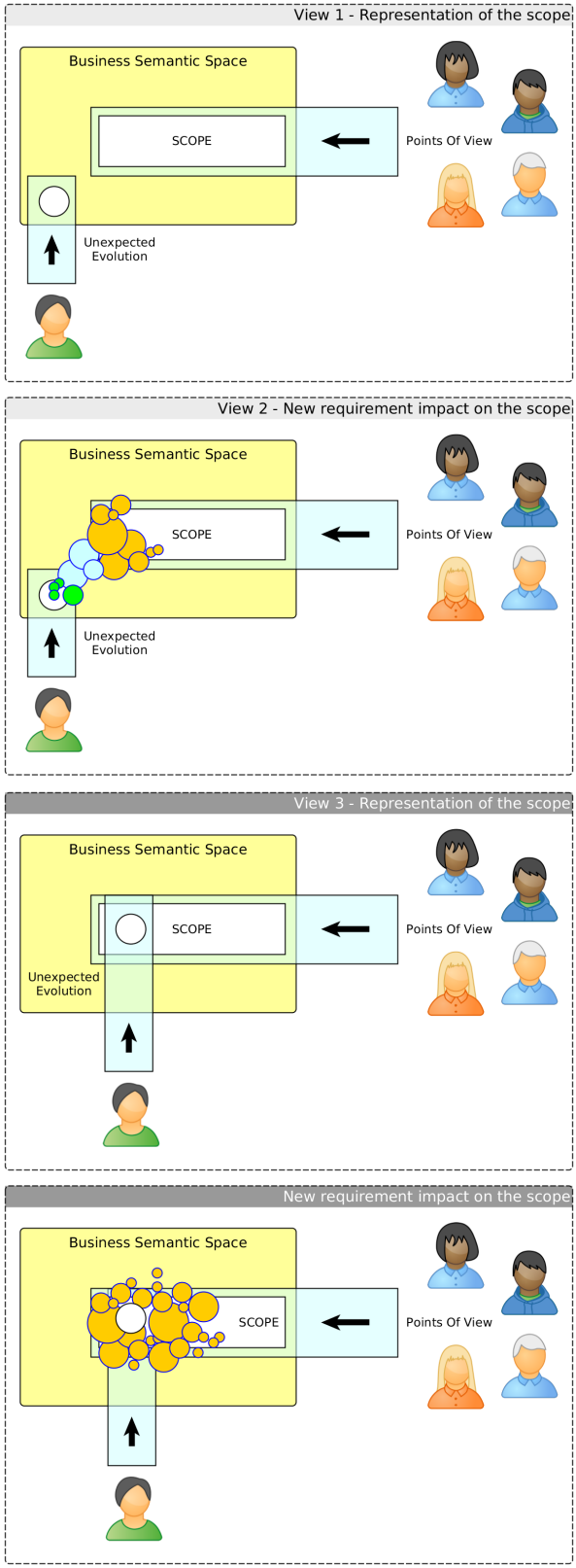

Figure 3: Evolution of the scope of an application

2.3.1 Design Patterns

The object-oriented design patterns [11] can be seen as *a set of empiric recipes* with three objectives:

1. Solve common design problems with common solutions;
2. Identify reusable components;
3. Try to solve the present design problems while easing the future evolutions (application extensibility).

For the first point, the approach is near from the mathematical theorem approach: under a definite set of conditions, we can apply a design pattern. If this set of conditions is not met, the design pattern should not be used, and in most cases, it could be harmful to use it because it introduces unnecessary complexity that will be paid during the maintenance and evolution phases[9].

For the third point, things are more complicated because forecasting the "directions" the software will evolve to is a very difficult task. In most of the projects we saw, the capability of software engineers to properly anticipate, at design time, the directions for software evolutions is quite *low*. That means that, in the maintenance and evolution phases, redesign is often needed because of bad evolution forecasting options taken in the past.

This should not be a surprise for the software industry: it is very difficult for software engineers to anticipate changes in the business; indeed, it is not *their* job. Software engineers know how to make the best design with a certain *scope* but it is almost impossible to design a system that will be robust to any kind of scope change. It would be better if the software industry could guarantee a set of technologies that support any kind of business change easily.

The promise of design patterns was to be able to ease future maintenance and software extensibility, often by trying to reduce coupling[10]. It is obvious that the objective was not fully achieved.

Perhaps, this is due to the fact that some crucial books [11] were published *too soon*, before the software industry had some real experience on what it was to implement big enterprise software with OOP/RDBMS[11] technology

2.3.2 Software Architecture Concerns

Software architecture [4, 32, 30, 1, 8] is at the heart of a very large literature. By defining cautiously the architecture of a software, software architects can limit the impact of change, for instance a database change or a web service signature evolution. This discipline is a set of rules to identify big blocks inside the application and link them together (in layers and in components).

[9] Pattern misuse is still a great problem in the legacy code, because instead of easing the maintenance, it makes it harder. Some companies went too far in the pattern obsession by integrating pattern implementation in the yearly software engineer's objectives. This led to a lot of software troubles and massive unnecessary overcosts.

[10] Through the use of interfaces or virtual classes for instance.

[11] Relational DataBase Management System.

Those methods help to reduce some of the coupling that are more related to software component dependencies and separation of concerns.

But, when the core business model needs an evolution, several layers and several components, plus the database, still need to be updated. In terms of business code, the evolutions can be easier because the application architecture is good, but the required redesign will have to be performed anyway.

2.3.3 Refactoring Methods

A. Refactoring Because of a New Structural Use Case

Despite all efforts invested in design, most software professional know that a day always comes when a new requirement requires a structural (unplanned) change in the existing code. Because the business practices evolve, a new use case can change the way we look at the software.

Possibly, this new use case can require:

1. A redesign of at least one part of the software;
2. Changes at the database level that will require a *data migration* (from their original structure to the new one);
3. Software non-regression testing.

Indeed, for every software, many choices were done during the design phase. After some years, those choices appear as not being able to accommodate every change easily. The software needs *refactoring*.

For instance navigation choices are usually made taking into account the main use cases of the application. At a certain point, even the foreign keys of the database are designed in a way to ease the chosen navigation. Those choices make alternate navigations harder. In case of an evolution that would require a reverse navigation, the impact on the existing application (code and data) will be huge.

B. Refactoring Because of Software Fragility

Another common use case for refactoring is software instability.

When a piece of software was maintained once too many, bugs can appear as side effects of minor modifications. The code has become very fragile with time. It manages lots of cases and making an evolution in it has become very risky. When the fixed bugs in code generate more bugs, it is time for refactoring [10].

Refactoring need is typically the direct consequence of several problems:

a) Coupling;

b) Quick and dirty maintenance and evolutions;

c) Time;

d) Many software engineers maintaining the same code over the years.

At a certain time, when the maintenance and evolution is becoming too expensive, there is a cost to pay just to reimplement the same features differently in order to reduce bugs and problems.

Refactoring is hard, painful and dangerous because:

a) It needs to preserve the exact same functionality inside the software;

b) It needs to enhance software maintainability (to avoid respending money on that particular piece of software);

c) It needs to make the new cleaner design coexist with bad old designs (which implies complex technical migration and cohabitation plans);

d) It needs to maintain data integrity and semantics (when the database is also touched by refactoring);

e) It calls for a massive non regression testing campaign to ensure that all is right.

Sometimes, it is too expensive to refactor, especially when software engineers used anti-patterns [3]. In those situations, both solutions are bad: either the application is refactored, it is dangerous and it costs a lot; or it is rewritten, it is extremely dangerous and it costs even more.

This explains why so many legacy applications are not refactored nor rewritten.

C. A Work-Around

Refactoring is painful, risky, costly, but necessary. The problem is that it is and remains a mandatory work-around. What is conceptually problematic is that the software industry has *no proper way* to realize enterprise software that would require *no refactoring* one day.

2.3.4 Object To Relational Mapping

For many years now, a lot of software engineers are using ORM systems to store objects inside relational databases. The promise of the ORM is double:

1. Abstracting the software engineer from the relational mapping (through code generation);
2. Manage conceptual evolutions by regenerating easily mapping code.

Actually, this works quite well for simple applications with simple queries and no structural evolutions. But for most enterprise software, an ORM system cannot cover 100% of the database accesses[12]. In that case, it becomes really problematic to use the ORM sometimes and manual storage management some other time[13].

Indeed the productivity of ORM system uses seems good in the development phase but it is more questionable during the maintenance phase, especially in case of structural changes. We could argue that the ORM system use enables to deliver quickly by pushing hidden costs in the maintenance and evolution phases.

[12] By experience, we could say that at least 20 to 40% of the database accesses cannot be dealt with the ORM helpers.

[13] Considering that most ORM systems are doing memory cache, the use of an ORM system in enterprise software can quickly become tricky.

We can note that there are some various philosophy in the industry about the use of ORM systems.

Even if it is quite complex to quantify, we can see the general tendencies in the market:

- Enterprise softwares realized in IT service mode for specific companies are generally using an open source ORM;
- Enterprise softwares realized by software companies for multiple customers generally avoid the use of an ORM (or are using their own product).

A part of the industry, near from the vendors, are pushing for the use of ORM systems in domain-driven applications [8], their main argument being productivity. Some authors even pretend that ORM data objects can be considered as the *business objects* [22], whereas in the domain-driven approach they are only the *data layer*, i.e. a kind of adapter between the business classes and the database.

Even if the idea is interesting because the question of "distance" between the concept and the code is a recurrent question of software, as we will see in part 5.2, we believe ORM systems can help in some limited cases, but not really for enterprise applications that have a real functional complexity. In other terms, if the application needs mainly CRUD[14] requests, the ORM system is very adapted; if the application requires more evolved data accesses (complex queries, TP[15] and batch), the ORM system use should be questioned.

ORM systems surely help to make certain evolutions inside enterprise applications but, once more, it automates painful code that needs to be there to manage the coupling attached to the OOP/RDBMS technologies.

2.3.5 Conclusion

The software industry created, with time, a lot of recipes and work-arounds to ease the maintenance and evolution phases.

But we must realize that existing code is not easy to maintain because, from day one, we are *coupling tightly* things together. Even if our design choices are relevant from the very beginning of the project, we cannot but create tightly-coupled code and database.

We could say that the software industry is used to create "*hard*-software" instead of searching for real "*soft*-software".

2.4 Time Management in Enterprise Software

2.4.1 Current Practices

Many changes in the applications come from changes in regulation or laws. Those changes are, most often, time-based changes[16].

In a lot of enterprise software, many structural changes are time-dated and contextualize rules with time. Old data must be governed by old business rules and new data (possibly with new structures) by new business rules.

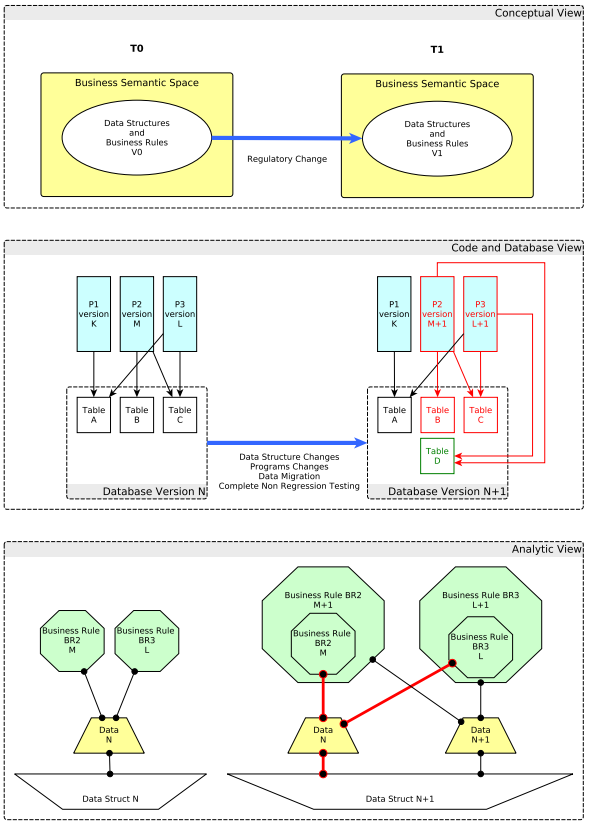

Figure 4: Evolution management in enterprise software

Let us consider Fig. 4.

By evolving the semantic space from `V0` to `V1`, we had to create a new table `D` and modify `P2` and `P3` programs.

The current development process will push us:

a) To migrate the old data (stored in version `N` structures) into version `N+1` structures;

b) The program evolutions will make `P2` and `P3` support version `N+1` of data structures.

The most common approach in the software industry is to create a data structure that will be a super set of all versions of data structures and to have programs managing different business rules depending on time.

This approach generates multiple levels of coupling between code and data. Let us consider the analytic view of Fig. 4. We take the hypothesis that `P2` and `P3` implement two business rules named respectively `BR2` and `BR3`.

On top of the coupling we identified in part 2.1, we are generating a *version-based coupling* or *temporal coupling*:

a) The `P2` and `P3` programs must evolve in version (respectively `M+1` and `L+1`) and so are their business rules `BR2` and `BR3`;

b) The data version `N` in structure version `N` is migrated to structure version `N+1`; We can note that semantically the data is still old version `N` data;

[14] Create, Retrieve, Update, Delete.

[15] Transactional Processing.

[16] For instance, in accounting systems, most migrations are done after closing, and the data migration of old data is not really possible (nor even meaningful).

c) `BR2` version `M+1` must contain `BR2` version `M` because this old version of the business rule must work on old data, even if the old data was migrated to the new `N+1` version (and the same for `P3` and `BR3`).

2.4.2 Temporal Couplings

All that gives us the following new temporal coupling (in red in the Analytic View of Fig. 4):

a) Data version `N` is coupled with structure version `N+1`;

b) Old code (`BR2` version `M` and `BR3` version `L`) is modified to manage old data stored on new structures (version `N+1`);

c) New code (`BR2` version `M+1` and `BR3` version `L+1`) must manage new and old data stored in version `N+1`.

Those couplings are really strange because the real need for evolution is that version `M+1` of `BR2` is managing data version `N+1` on data structures in version `N+1` and the same for `BR3`[17].

So why are we doing those complex, risky and costly migrations?

Actually, considering our ways of storing information, it is not (easily) possible to version the storage structure[18].

So, when a new requirement implies an evolution of the storage structure, this storage has to become the new storage *for all data of the same kind*: old ones, that potentially will never evolve anymore and do not need the structure modification, and new ones that require the structure modification.

For sure, as long as all data (old ones and new ones) are located inside the same table, there should be only one version of program to manage them, whatever their age.

We can imagine in what state are the data and the code after tens of regulatory changes: they have to become *unmaintainable*.

2.4.3 Time Management Consequences

This way of managing evolutions in software and data implies blocking progressively the system. After some years, it becomes extremely dangerous to dare a major evolution in the code because it may brake something or on current business rules, or on past business rules applying to past migrated data. Moreover, each new regulatory change can bring a real risk for the system to generate unexpected behaviors or bugs.

The core issues appear to us as dual:

a) The RDBMS use implies very complex, risky and costly data migration processes;

b) The absence of shared *timelining* software concepts and best practices for software engineering.

For instance, an application business rule really need to evolve if and only if the new rule applies to all data inside the database. If the rule applies only to recent data and the old rule still applies to old data, then the new rule cannot really be conceptually considered as an evolution of the previous rule.

In other words, the new rule has its domain of application in recent data, domain that is disjoint from the old data set that is the domain of application of the old business rule. By declaring *a priori* that only one rule should manage all cases, we fusion two functions that do not have the same domains of application. Another obvious way to solve the same problem is to have two rules that could declare themselves not applicable on the data they cannot manage.

2.5 The Cost of Coupling

2.5.1 The "Glue Effect"

Our software practices generate tight coupling inside the enterprise software, both at the code level and at the database level.

Those tight coupling, in the maintenance and evolution phase, make all evolution harder and harder with time. The design of the application and database cannot evolve easily, especially in contexts where time and budget are short.

It is surprising that the shareholders of IT projects (users, functional experts, software engineers, architects, managers, executives) seem to accept this continuous slowing down in the rhythm of evolutions and this continuous cost increase of any evolution[19].

We will say that the maintenance and evolution phases are demonstrating a *Glue Effect*: progressively, the enterprise software seems "glued" what prevents every significant change to happen in reasonable time and money. The software industry seems to live with both the problem and the work-arounds. Surprisingly, in the software world, it seems accepted that *software is as hard to evolve as concrete plants*.

Tight coupling is the material of *technical debt* [23, 25, 24]: the technical debt is the overcost that we must pay to develop a feature, compared to what we would have paid for the same feature in a project mode i.e. without code legacy constraints.

In systems, the Glue Effect can be explained by the generation of technical debt: with time, we should invest more and more money and effort to realize the same function.

2.5.2 The Huge Cost of Coupling

The software industry is focused for years on the rapid application development[20] but does not take into account the maintenance process and its costs.

If we analyze the cost of function point [27] after the initial deliveries, we realize that the cost of the function point is increasing with time. After a decade or more, we can see many applications that will have, for a constant budget, a number of function points developed per year that will tend to zero[21].

[17] For instance, in accounting, in banks, tax collections, insurances, pensions, aerospace maintenance, etc., past data do not need to be migrated because they will change no more.

[18] We can see in some huge accounting systems this versioning of storage with databases that are dedicated to manage specific structure for a specific set of years.

[19] Considering constant manpower.

[20] RAD method was a sample of this preoccupation. Agile methods are also focused on quick delivery of applications.

[21] Which correspond to a price per function point that tends to infinity.

If we consider that an application will last 10 years in maintenance after 2 years of project (build), the total cost of maintenance is quite often far superior than the project cost. Thus, the productivity during the maintenance and evolutions phase is of the utmost importance, budget-wise.

Moreover, in terms of company agility and capability for a company to accommodate business or regulatory changes, the productivity of the maintenance and evolution phase is crucial: If the company had to face issues adapting its IT systems, it could risk to be late compared to the competition. This process is also illustrated in administrative enterprise software that cannot evolve easily and that prevent any major process transformation. This problem seems to us as one of the biggest problems in enterprise software. It costs billions of dollars to states and companies.

Philosophically, even if we cannot pretend to prove the following statement, the reduction in the global productivity rate that the economists saw in the last few years may be partially due to the complexity of maintenance and evolution phases of the enterprise software in companies and administrations. Most companies and administrations live their IT systems as problems due to the Gordian Knot: invest massively for small risky evolutions or invest even more massively for a very very risky rewriting. This causes a freeze in process evolutions and a very huge difficulty to adapt the business processes to new constraints and challenges.

For sure, IT service companies have a certain interest to maintain high costs of maintenance and evolutions. During the life cycle of applications, more money is generated from the maintenance and evolution phase than during the project phases where market prices are low due to high competition. So, when IT professionals have a certain interest not to be technically efficient in certain phases of projects, we can understand that the industry is not very keen on searching for new effective solutions.

2.5.3 Conclusion

As a conclusion, we can say that, structurally, the core technology we use (object-oriented programming and relational databases) forces us to generate tight coupling and technical debt from the very beginning of the application construction[22].

The graph-oriented approach will enable the software industry to reduce drastically the technical debt. For that, we must adopt a new way of modeling and a new way of programming. We will call this approach the *graph-oriented programming paradigm*.

[22]This is often seen in projects when the development phases phases start before the design is done or ended. The project has to cope with legacy code that software engineers tend to reuse whereas there is not a single line of code running in production. Thus, some projects may get stuck in legacy code from the very beginning of the project and even before completing the software development.

Feature	Semantic (RDF) Database	Relational Database	Graph Database
Nodes	Yes	Tables	Yes
Node types	No (String)	Yes	Yes
Node typed attributes	No (String)	Yes	Yes
Relationships	Yes	Foreign keys	Yes
Relationship types	No (String)	No	Yes
Relationship typed attributes	No (String)	Yes	Yes
Query language	SparQL	SQL	graph-oriented query language

Table 1: Comparison between database types

3 The Attributed Directed Graph Databases, a Game Changer

The creation of attributed directed graph databases[23] in the software world is an event of the greatest importance. Even if, at the time this paper is being written, the graph databases are mostly used in the Big Data world, their use in enterprise software will revolution our very way of thinking about software.

By (attributed directed) graph databases, we mean databases that have the following characteristics:

1. They manage node[24] and relationships typed instances[25];
2. Node and relationship types can have a variable set of typed attributes;
3. They propose some kind of graph query language[26].

This way of storing data brings a unknown softness in data structures (compared to relational databases) because they manage links *outside* the node itself, which enables the evolution of links without changing the structure of the nodes (contrary to relational database foreign key concept). This enables the evolution of data structure through evolutions of relationship types and node types independently.

Indeed, the attributed directed graph database as we know it today is an intermediate between relational databases and semantic databases as shown in table 1.

The primary objective of this article is to explain what programming paradigm can be used with graph database technology to have *the same softness in the software than we have inside the data structures*.

[23]In the rest of the document, we will speak about "graph databases" instead of using the term "attributed directed graph database".

[24]We can also name nodes vertices and relationships edges.

[25]Or some kind of typing through labeling in the Neo4j database case.

[26]For instance Cypher in the Neo4j case or a SQL extension in the OrientDB case.

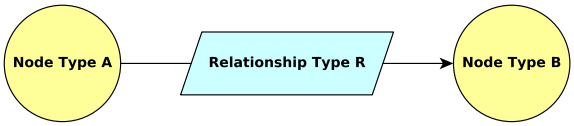

Figure 5: Node and relationship types

Figure 6: Node and relationship types with attributes

4 Introduction to Graph-Oriented Modeling

In order to illustrate our speech, we have to introduce, in this article, a domain specific modeling (DSM) language to be able to represent the graph concepts in a consistent way [29].

Going through the article, we will progressively enrich the meta model of this DSM language to be able to graphically illustrate the various features of the graph-oriented programming paradigm.

4.1 Representing Node and Relationship Types

4.1.1 Basic Artifacts

First of all, we need to represent node types and relationship types. The graphical conventions that we will take are shown in Fig. 5: yellow circles for node types and blue parallelograms for relationship types. Note that the relationship is oriented; this enables us to designate a *source* node (`Node Type A`) and a *target* node (`Node Type B`).

With the artifacts defined, the Fig. 5 builds the graph-oriented structural view.

4.1.2 Why Not Use the UML Class Diagram?

The UML notation (class diagram) is not adapted to the representation of graph structure (representation of node and relationship types) because UML does not materialize the relationship types as plain entities that can contain attributes. In UML, the relationships are represented through aggregation, composition, extension or association lines. The Fig. 6 shows node and relationship types with attributes, which is impossible to represent in UML[27].

The Fig. 7 proposes a comparison between the two modes of thinking: the object-oriented one (represented by the UML view) and the graph-oriented one.

We can discover in this figure that UML does not specify the semantics of the relationships between object types (classes).

[27] This is not totally true because we could use stereotypes, but as we will see, there is a great interest to define a specific graph-oriented modeling approach.

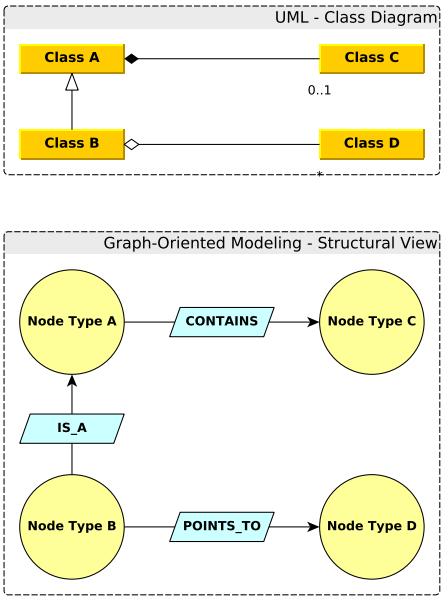

Figure 7: Comparison between UML class diagram and graph-oriented modeling structural view

Through the extension, aggregation and composition symbols, UML proposes some kind of *generic graphical syntax* for three of the four traditional relationships (displayed in Fig. 7). This generic syntax is not an issue because, in an object-oriented world, structural relationships are implemented (in the code) in a non ambiguous way.

When modeling the graph-oriented corresponding diagram, we have to make semantic choices and name the relationship types also. The names we took for relationship types (`CONTAINS`, `IS_A`, `POINTS_TO`) are choices that are coming from our way of considering the system we are modeling, but our choices could be different.

Actually, from the very beginning of the modeling of an application, the naming of the relationship types introduces semantic questions that are not present in UML and object-oriented modeling. We will see, in the coming parts, that some practices of the graph-oriented modeling are quite near from semantic design [33].

4.1.3 No Structural Relationships

UML considers aggregation, composition and extension as being *structural relationships*.

In graph-oriented modeling, we have no distinction of the kind: relationship types can be structural *in their semantics*, i.e. relatively to their meaning. However, the graph-oriented DSM language that we propose to consider, does not provide any way to tag a relationship type as structural, or to distinguish so-called structural relationships from the others.

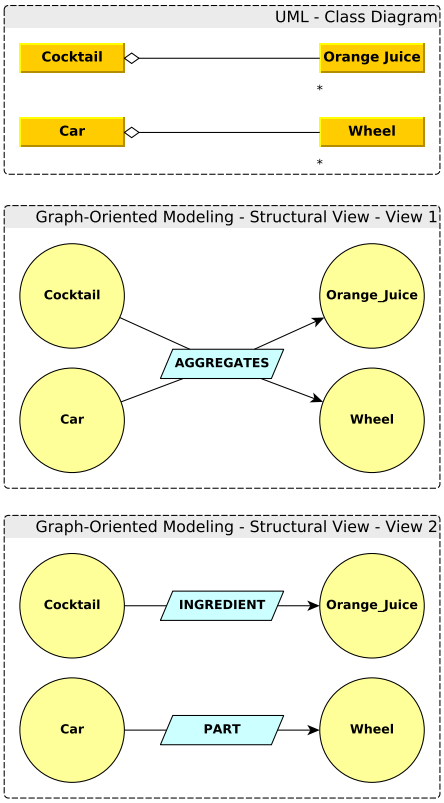

Figure 8: Representing aggregation in graph-oriented modeling

4.1.4 Representing the UML Aggregation and Composition Links

To go further on the treatment of object-oriented aggregation and composition links, let's consider Fig. 8 that proposes a sample of semantic ambiguity in UML: the cocktail and the car sample.

In UML, the use of aggregation is normal in this case (class diagram on top on Fig. 8): Neither the `Orange_Juice` nor the `Wheel` can be considered as a composed class of respectively `Cocktail` and `Car` classes because `Cocktail` and `Car` classes have their own life cycle inside the application. In UML, as we are quite near from the object-orented implementation, we can reuse the same aggregation symbol in the design because at the end, the code will not materialize the aggregation link as a separate concept but as a structural relation (a member of the class `Cocktail` will be an instance of `Orange_Juice` and a member of the class `Car` will be an array or a vector of some sort of instances of the class `Wheel`).

In the View 1 of Fig. 8 (box in the middle of the figure), we can see what would be the graph-oriented equivalent design if we had use the same relationship type `AGGREGATES` to represent the two UML aggregation symbol. This modeling appears as quite ambiguous, because `Cocktail` does not `AGGREGATES` `Wheel` nor `Car` `Orange_Juice`.

In UML, the notation is also ambiguous because we cannot express the fact that once linked, the `Cocktail` and the `Orange_Juice` cannot be *separated*, whereas the `Wheel` can be separated from the `Car`. As the aggregation link is considered (a) structural and (b) generic, the code will have to deal with this specificity that cannot be expressed at the modeling level.

The View 2 of Fig. 8 introduces a different graph-oriented modeling of the aggregation relationship. With two relationship types to address both uses of the UML aggregation symbol, we add a semantic specificity to each link.

This will enable the code to consider that the `INGREDIENT` relationship is behaving differently from the `PART` relationship. For instance, we will be able to express that, in an instance of `Cocktail` containing `Orange_Juice`, we cannot remove the `Orange_Juice` once it is melted.

The last thing that we will note about the Fig. 8 is that our semantic enrichment did not really go through a new *verb* definition (like `AGGREGATES`) but through a characterization of *features*. In UML, the link being named "aggregation" makes the following wording possible: a `Car` aggregates a `Wheel`, which look like a semantic RDF-like triple. Actually, the graph-oriented modeling equivalent would make the following wording possible: a `Wheel` is a `Car` `PART`.

This nuance is not a general property because, sometimes the relationship type in graph-oriented modeling will express a verb and, some other times it will express a characterization of a feature or something else. By having the means to detail relationship types, we have, in graph-oriented modeling, expressive capabilities that go far beyond the way we usually model in object-oriented mode.

That is a fact that the UML aggregation/composition link generally hides semantics that graph-oriented modeling exhibits. If we were to provide an explanation for this fact, we would say that this syntax is a direct projection of the implementation concerns.

We can note that, on the contrary, in graph-oriented modeling, we currently have no real idea on how the implementation will be done. Designing without the implementation constraints in mind is, for us, quite a progress, as we will see in the rest of the article.

4.1.5 Representing the UML Specialization Link

We can propose the same kind of analysis for the specialization link that is available in the object-oriented approach. The Fig. 9 shows a standard graph relationship between concepts instead of using a generic specialization relationship.

We will not enter, in this article, in the discussions about the consequences of specialization on methods (we will come back to the treatment side of graph-oriented programming afterwards)), such as the method inheritance or the "deadly diamond of death" [28], but we will just note that the capability of representing extension by a relationship type opens new perspectives.

For instance, in Fig. 9, the View 1 is equivalent to the UML view, except that the `IS_A` relationship may not be used in all the cases where the extension UML relationship is generally used.

On the contrary, expressing specialization with typed relationships enables to enter new semantic possibilities:

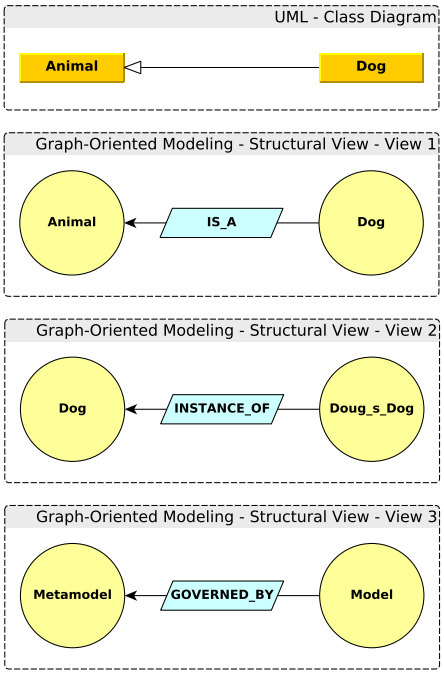

Figure 9: Representing specialization in graph-oriented modeling

1. In the View 2 of Fig. 9, the `INSTANCE_OF` relationship makes a bridge between instances and type, which is rather unusual in enterprise software (but is quite usual in semantic design);

2. In the View 3 of Fig. 9, the `GOVERNED_BY` relationship enables to manage, inside the same storage system, the model and its metamodel and to perform operations at both conceptuel levels.

As in graph-oriented modeling, there is no structural relationship (in the object-oriented meaning). We have to make a semantic choice to characterize the specialization link. One consequence of that fact is that it opens new perspectives because we can manipulate several level of abstractions at the same time.

Once again, in a classic object-oriented approach, the specialization link is quite near from the implementation. By creating a class hierarchy, we intend to have the inheritance of attributes and methods. This concern often tweaks the design with implementation concerns.

4.1.6 Comments on Relationship Type Constraints

The relationship artifact that we introduced has the same property as its attributed directed graph database equivalent:

a) An instance of relationship is existing if and only if a source and target nodes exist;

b) If the source or the target node or both disappear, the relationship instance disappears;

Relationship	Source Node Type	Target Node Type	Cardinality constraint
INGREDIENT	Cocktail	Orange_Juice and other ingredients	One instance per target type
PART	Car	Wheel or other part	4 instances of PART relationship per standard car

Table 2: Sample of relationship constraints table

c) In terms of types, the relationship types can connect several node types; we can also have relationships that take `*ANY_TYPE*`[28] as their source node type and/or `*ANY_TYPE*` as their destination node type.

In graph-oriented modeling, we will have the requirement of representing constraints on relationships. The topic of constraints is a common complex problem in modeling and we will not enter in details in it. We will mention the two constraints that we find important for most enterprise software modeling:

1. Cardinality constraints;

2. Source and destination node type constraints.

For instance, considering Fig. 8 View 2, we can define the table 2 that describe the constraints that we have relatively to cardinality and node types.

For sure, the semantics of the constraints should be detailed in the case of the implementation of a graph-oriented modeler.

Though the expression of those basic constraints, we can express various subtle things such as:

a) When the relationship type is limited to a certain set of source and destination node types, it becomes a semantically non ambiguous expressive way of describing the business;

b) When the relationship type points to `*ANY_TYPE*` or is pointed by `*ANY_TYPE*`, we can imagine that the relationship type is a *bridge* between *domains* and is used in some kind of extensibility mechanism (see part 7.4).

Some modeling languages are proposing many tools or are dedicated to constraint modeling [31, 14]. However, by experience, we consider that too detailed constraint expression in modeling can be dangerous because it pushes to express *parts* of business rules in the model while the other parts must be expressed textually. This split of semantic information is one source of trouble in the implementation phase. For this reason, we always saw a quite low adoption of constraints languages in the design phases of projects.

Actually, cardinality constraints and relation type connectivity constraints are quite straightforward, in the sense that they generally do not express parts of business rules. It is, at the

[28] We will use the notation of a text between stars when we refer to generic concepts (relationship or node types).

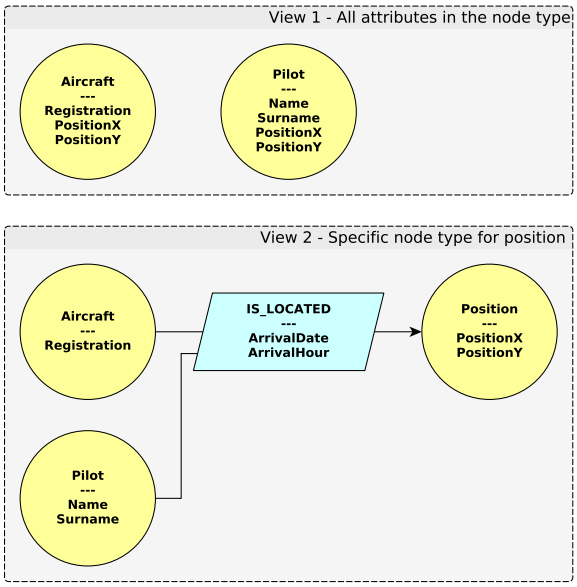

Figure 10: Typed Attributes as Node Types

same time, the minimum constraints that should be expressed and the maximum before entering too deeply in business rules.

We can mention other approaches like business rule modeling languages [**?**] that can be considered as an extension of constraint modeling. Once again, they seem not to be widely used. In graph-oriented programming, we will prefer to deal with business rules through the notion of graph transformation (see section 6).

To conclude on this point, we will mention that current attributed graph databases in the market do not propose simple ways on creating constraints on relationship types[29].

4.1.7 Node Types or Typed Attributes

One of the consequences of being near from semantic design is that graph-oriented modeling is facing the same questions as in object-orientation about the attributes. When you store typed attributes inside a node type, the question raises to group those attributes as a separate node type.

Let us consider Fig. 10. The View 1 is proposing to represent two business concepts `Aircraft` and `Pilot`, each of them containing position attributes `PositionX` and `PositionY`. The View 2 defines a new concept `Position` grouping those two attributes `PositionX` and `PositionY`, and a new relationship type `IS_LOCATED` having also two attributes `ArrivalDate` and `ArrivalHour`.

The View 2 enables to determine easily that both the `Aircraft` and the `Pilot` are at the same position. If some geolocation library is available, we can imagine to search for all node instances of type `Position` and to go back the `IS_LOCATED` relationship instances to find all instances of `Aircraft` and the `Pilot` that are in a certain location or near. The relationship `IS_LOCATED` is introducing a kind of *bridge* between the `Aircraft` and `Pilot` business concepts and the *semantic space* of `Location`.

We can mention another sample of that attribute externalization phenomenon coming from an industrial project that needed specific color management. The externalization of the color attribute in a specific node type `Color` led to the building of a lot of functionality around color management, and to think differently about color and product relationships: it defined a *color semantic space* that possessed its own dedicated treatments. In the design phase, it was possible to separate the color management from the rules that were attaching the `Color` to the industrial `Product` is was an attribute of.

For sure, all this is a very classic concern of software design and we can use attribute externalization in object-oriented programming also. But graph-oriented modeling manages this externalization far better by introducing a relationship type that bridges two semantic universes. The relationship types being non-structural, the reuse of the attribute-based node types becomes much easier in graph-oriented programming than in object-oriented programming.

4.2 Introducing the Domain Concept

A direct consequence of the disappearance of structural relationships is that we can work of graph-oriented data at several abstraction levels. As we just saw in Fig. 10, through the use of some relationships, we can also define different semantic spaces. In a way, the abstraction between the relationship linking the model elements to their metamodel definition (see Fig. 9 View 3) can also be seen as a *bridge* between two semantic spaces[30].

We propose to name those semantic spaces *domains*, because in the software industry, the notion of business domain is widely used [8][31].

During the graph-oriented modeling phase, the idea will be to analyze the business and to group node and relationship types by domains. We can imagine a lot of kinds of domains: business domain, technical domain, utility domain, metamodel domain, etc.

The Fig. 11 shows a multi-domain approach through a sample from the aerospace maintenance. Domains are containing node and relationship types.

Some relationship types can be *bridges* or *inter-domain* relationship types (partially represented with a double arrow in Fig. 11). This point is very important because this inter-domain connectivity is one of the crucial sources of software extensibility in graph-oriented programming (see part 7.4).

4.3 Overview of Some Graph-Oriented Modeling Practices

In this part, we will list some practices that emerge when we are doing graph-oriented modeling in the context of enterprise

[29] Some of them do not have real constraints on attributes neither.

[30] Some RDF-based graph databases such as AllegroGraph are clearly adapted for the definition and exploitation of semantic links between various semantic spaces.

[31] In some markets such as France, the notion of (reusable) *business objects* in enterprise software can be dated to the beginning of the 90's.

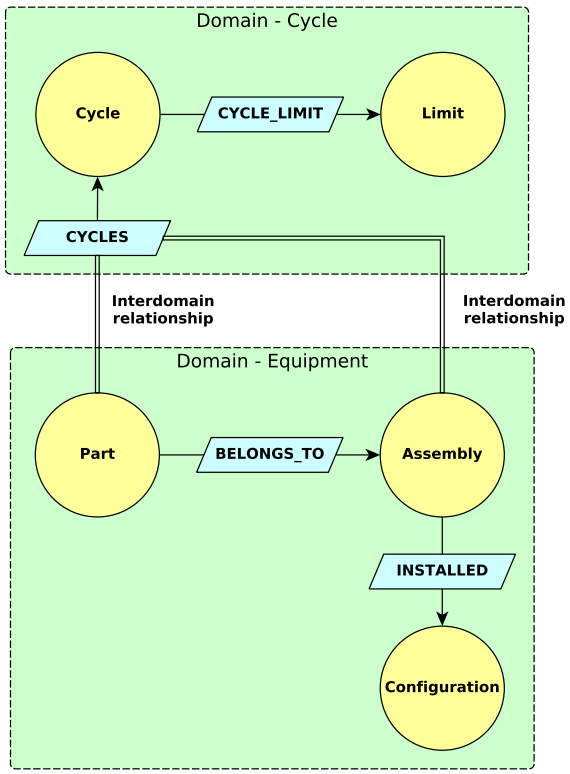

Figure 11: Domain concept illustration and inter-domain relationships

software.

4.3.1 The Business Functional Analysis in UML

Theoretically, in projects using an object-oriented approach, we should have several levels of modeling:

1. The use cases;
2. The functional analysis model;
3. The design model;
4. The database model.

In most object-oriented development methods, like [16, 6], after the use case definition phase (or in parallel), the method proposes a phase of *business functional analysis* to perform before entering into the design phase. During this analysis phase, the objective is to realize the modeling of the semantics of the business domain, identifying business concepts, relationships between those concepts and cardinality.

In this phase, the use of the specialization, aggregation and composition links is often not recommended, because it may introduce implementation concerns inside the functional analysis. The functional analysis phase is quite often a real important phase of good enterprise software construction[32].

[32]The analysis diagrams sometimes look like "concept maps".

This phase is not widely practiced in the software industry because most software engineers consider it as being an unnecessary effort compared to the design phase. But rushing into design without a proper business analysis may introduce a certain confusion between the business semantics and the business implementation.

4.3.2 The Progressive Abandon of UML

For more than a decade, we have seen a progressive discontinuing use of the design phase (in UML) in software projects, which makes it even harder because instead of rushing directly in the design, software engineers directly rush into the code.

Agile methods sometimes push for those "code-first think-after" behaviors.

One other reason may be the complexity of web application design that makes UML not really adapted for the Internet: the Web technologies are not fully object-oriented, they use several programming languages (HTML, JavaScript, CSS and a Java or .Net server[33]) and paradigms (document-based, functional, object-oriented, and so on) and various client-server approaches.

This implies that, in a lot of software projects, only database models can be found (mostly because they can be generated *a posteriori*).

4.3.3 Reintroducing Modeling

In a certain way, UML did not fulfill all its promises because:

a) It is a set of disconnected symbolic languages (one per diagram) and it is quite complex to ensure specification completeness with UML;

b) It is not really adapted to Web application design (nor to server side application design); It can be used in those cases but with specific conventions and work-arounds;

c) It is not a method but a language, which means that it cannot be used off-the-shelf but within a certain method that most project do not have the time to set-up.

However, the absence of modeling (being in functional analysis or design) do not make problems disappear. Modeling is still required in enterprise software and not doing it leads to great inefficiency in the maintenance and evolution phase[34].

4.3.4 About Graph-Oriented Modeling

Graph-oriented modeling fulfills the dream of object-oriented designers because it is at the same time:

a) A semantic analysis that is more expressive than the object-oriented functional analysis phase;

[33]In some Web projects, there can be more esoteric languages or syntax designed to be JavaScript-based domain-specific languages or JavaScript generators.

[34]Maintaining a big software without design documents is like trying to find its way in the metro with two closed eyes.

b) A design action because what is designed will be implemented in a straightforward way (down to the database).

Moreover, graph-oriented modeling does not presume about implementation and the certain views created with the DSM that we propose can be shown to non-software users, provided they are able to understand conceptual diagrams.

Graph-oriented modeling is a huge step ahead at the same time for software engineers because it enables one modeling for several phases, and at the same time it is a huge step ahead for users because they will be able to understand IT documents and models.

However, we will insist on two very important facts:

1. Models *must* be used in all phases, because the negative side of all graph-oriented modeling advantages is that the application can become very complex, which makes it mandatory to maintain a map;

2. Models must describe the *structure* of the business, i.e. the node and relationship *types*, and instance-based design must be restricted to complements to structural modeling, and must be considered as *samples* enabling to understand the structural model.

In particular, as an echo of the second point, we cannot imagine to design enterprise software with only instance-based views.

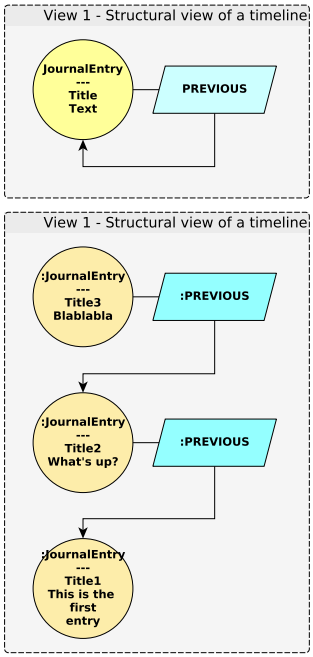

Figure 12: Instances

4.3.5 Instance-Level Modeling

Some graph databases software vendors propose a modeling phase that is only based on node and relationships instances view. This position seems to us as not sufficient to build large enterprise software. Indeed, the corresponding UML assertion would be that is is possible to design big software with just collaboration diagrams.

For sure, instance-based diagrams are very important, especially when the various instances of the same types do not play the same role in the represented figure (but that is a common topic in modeling).

The Fig. 12 shows the difference between a structural view (View 1) and an instance view (View 2). It is easier to understand how the `PREVIOUS` relationship works when we see the instances (noted with a leading "`:`" in the figure).

Note that the graph-oriented modeling approach enables us to link together instances and types like shown in Fig. 13.

This way of representing can bring some confusions and so, if it is used, it should be with strict naming conventions on relationships. In the figure, we named with surrounding "$\star$", the `INSTANCE_OF` relationship types and we used a suffix with the name of the node type. The stars indicate a very special case of relationship type (link from the instance to its type) and the suffix enables not to reuse those links for all node types.

Even if the Fig. 13 can be considered as a style exercise, it shows that, in graph-oriented modeling, the delimitation between instances and types is sometimes not so clear (which is also a characteristic of semantic design [33]).

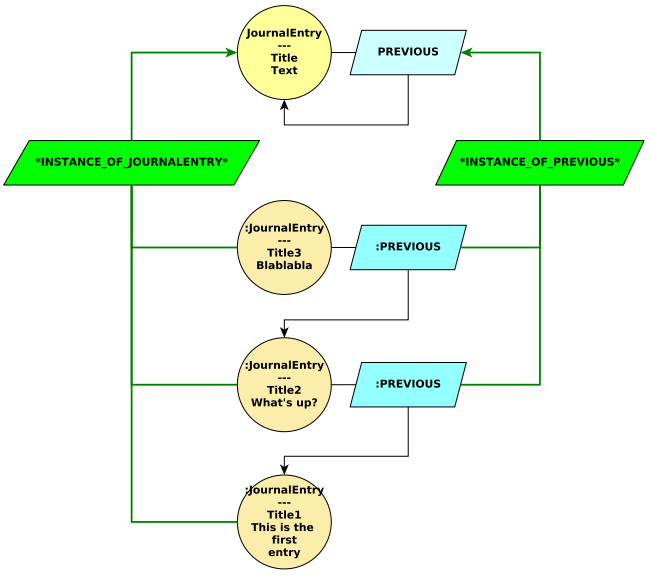

Figure 13: Instances linked to types

Considering this tweak, the modeling tool could be limited to the management of one single type of view, or it could propose structural and instance views.

4.3.6 One Model, Multiple Views

The last point that we will mention is the need to navigate inside the model. This part is very important in the way that, the graph representing the business at the core of an enterprise software, will not be representable in a single view. Like in UML, we need many views with the capability of showing node and relationship types on many incomplete viewpoints.

Once the views are realized, the modeling tool should offer the capability to navigate *in the graph model* composed by the union of all views.

This enables an iterative process to go from individual views to a model exploration and from the model exploration to the individual views. This process seems to us as crucial in graph-oriented modeling.

The reason is, in a large software, the graph model cannot be grasped in one shot by a human mind. Even if we use domains to group entities, the exploration enables to center the model on a specific node type and the study of the relevance of all incoming and outgoing relationships, independently from the various views that represented probably various use cases of the application.

The Fig. 14 shows a generated model view centered in node type `A` and that aggregates the modeling constraints done in View 1 and View 2. From node type to node type, the model can be browsed in order to reach a global consistency[35].

4.4 First Conclusion On Graph-Oriented Modeling

Graph-oriented modeling is a big step ahead that enables to fusion many phases of the traditional object-oriented design-process. Moreover, it enables to define a common language between users and software engineers.

Graph-oriented modeling process coupled with graph-oriented programming (that we will detail in the coming sections) should accelerate enterprise software development while reducing the gap between the user and the software professional understandings.

However, we think a modeling tool must be used and maintained up to date, at least for the node and relationship types[36]. Moreover, this tool must provide graph-adapted tools (such as model exploration) to be able to solve semantic ambiguities and achieve global consistency.

Graph-oriented modeling is an iterative process that will, at a certain moment, converge to a robust model describing properly the business. In a certain way, this process may appear as much more iterative than the object-oriented design, because of the required identification of the proper relationship types. Graph-oriented modeling is integrating semantic design into enterprise software and enables many advantages that we will discover in the rest of this article.

[35] We can see those kinds of practices in enterprise architecture modeling.

[36] We will see more artifacts in the coming sections.

5 First Implementation Aspects

Even if we did not introduce all modeling concepts yet, it is time to get a first look on the implementation aspects.

5.1 Code Representation of Graph Concepts

In the previous sections, we saw basic modeling entities: node and relationship types. We have to clarify in what way those entities are represented inside the code itself.

The representation of node types and node relationships inside the code depends on the programming language that we use. We can define quite distinct representations of those entities depending on the fact that we are in an object-oriented language or a functional language. However, we think the spirit of graph-oriented programming can be quite near in all sorts of programming languages.

5.1.1 Node And Relationship Types

In the case of object-oriented languages, we must warn again about the fact that we cannot use the native aggregation/composition/extension links inside the code. Even if the language allows it, we must not use it or we would create the same dependencies as the ones exposed in part 2.1. For functional languages, it is the same. If we use a list of attributes to represent a node, we cannot use an attribute to contain a pointer to another node, but we have to use a dedicated relationship structure that is not coupled with the source and target node type representation.

This being said, we are going to define the minimum set of requirements that we need to ensure the proper working of a graph-oriented set of programs.

This minimum set is presented in table 3.

With the requirements described in table 3, we can create nodes and relationships[37], set a relationship to point to known nodes (source and destination).

5.1.2 The Need For a Memory Graph Structure

We need a way to represent *the graph in memory* as an opaque structure that will provide graph manipulation APIs. We will speak about this concept by using the term `Graph` (whatever the programming language that we are considering).

We can wonder why we need such a structure considering that each relationship has source and target IDs and that it is easy to rebuild the graph manually.

We see two reasons for that:

1. The first one is that manipulating graphs in memory is a tricky thing and that, for efficiency purposes, it is better to use some kind of memory graph management library, that would propose some convenient APIs such as the ones detailed in table 4;

[37] By nodes and relationships, we mean node and relationship type instances.

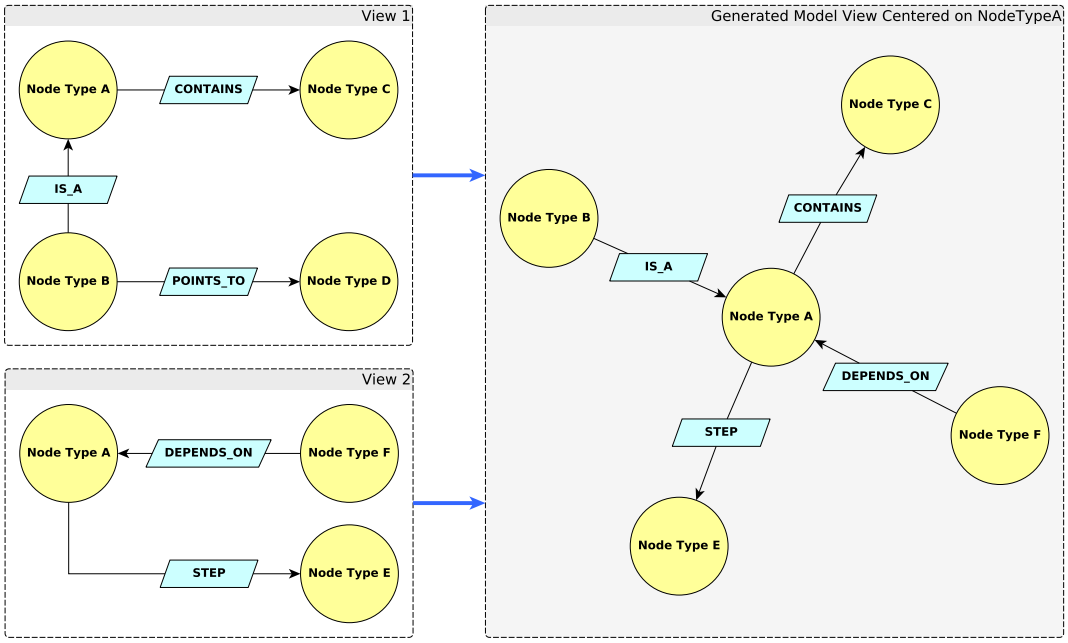

Figure 14: The generated model view centered on a node

Requirement	Sample in OO language	Sample in functional language
Programmatic representation of node type	Class	List/struct with typed members
Programmatic representation of relationship type	Class with a source node ID attribute and a target node ID attribute	List/struct with typed members with two members for the source and target node IDs
Programmatic representation of graph	There must be a class `Graph` enabling graph manipulation. The implementation does not really matter.	There must be a structure representing the graph and enabling graph manipulation. The implementation does not really matter.
Graph manipulation API	Methods on the `Graph` class	Functions acting on the graph structure

Table 3: Minimum set of requirements for a graph-oriented implementation

2. The second one will find all its meaning in the section 6; It is based on the fact that it can be very interesting to manipulate the graph *from the outside* of the graph structure itself.

As an echo of the second point, we can say that when implementing business rules in object-oriented programming, we have to navigate explicitly through the aggregations inside the objects. This generates a code that is completely linked to the structural class dependencies.

If we manipulate the (memory) graph from the outside, that means that we can consider it as a separate entity (`Graph`) that:

a) Can demonstrate certain configurations of nodes and relationships, what we will call *graph topology properties*;

b) Can propose ways to manipulate graph parts (i.e. nodes and relationships).

In graph-oriented programming, we can separate both concerns, when it is not possible in object-oriented programming. We will discover how powerful this approach can be in section 6.

5.1.3 Graph Manipulation API

The graph manipulation API must provide basic and advanced manipulation routines for graphs. The minimum requirement list for the graph manipulation API is provided in table 4.

For sure, we could probably imagine other features for the graph API but this list is a first set of tools that are at the same time very useful and commonly required.

In terms of programming, the concept is simple: each individual node and relationship is manipulated alone, while manipulations that require more than one element are performed in the graph structure itself through the graph manipulation API.

This way of proceeding has many advantages because there is no temptation to tightly couple things together. Actually, the way the (memory) graph is implemented internally has no real importance[38] on developments. We can imagine various implementations depending on the constraints the enterprise software is submitted to[39].

[38] We can imagine memory graph storage that do not store graphs as objects or structures but in more optimized ways.

[39] For instance, if the application needs to manipulate large sets of nodes in memory, the implementation of `Graph` could implement a kind of memory indexing mechanism.

#	Category	Requirement Description
01	Basic	Create graph from select query
02	Basic	Get graph root node (in the context where the root node has a meaning, we will come back on that point)
03	Basic	Add nodes and relationships inside the graph
04	Basic	Delete a node or a relationship inside the graph
05	Basic	Modify a node or a relationship inside the graph
06	Basic	Get nodes and relationships from the graph to access them in a object-oriented or functional way
07	Advanced	Assert a topology condition on the graph (returning `true` or `false`)
08	Advanced	Search for nodes and relationships with some criteria (such as per attribute value)
09	Advanced	Merge two different graphs
10	Advanced	Persist the graph
11	Advanced	Match a pattern in the graph
12	Advanced	Perform some other complex operations on graphs (for instance, for two graphs G_1 and G_2, create the graph $G_3 = G_1 \cap G_2$)

Table 4: Minimum set of requirements for a graph manipulation API

The requirement about the graph persistence (#10) should to be considered mainly in case the code manipulating the graph is located on the server side (like in a typical Web application). This service can be implemented at the graph level[40] and be a real help for the programmer.

We can note that we do not insist on graph queries, including traversals, except on requirements #01 and #09. For sure, we need a way to query the database and to create a graph with the results. But the benefits that graph databases bring to enterprise software go far beyond the powerful query languages, as we will see in section 6.

About querying the database and creating a graph with the result of a query, it is important to note that not all queries provide a graph: some provide graph parts but some may provide values for fields. Requirement #01 must be understood this way: if the result of a select query is a graph, then the graph manipulation API should enable to create a memory graph with it. The memory graph library will also manage *parts of graph* that are not mathematical graphs (for instance a solitary relationship pointing to node IDs that are not included in the graph, a node and its outgoing relationship that points to another node that is not is the graph, etc.). The graph notion that we will have at the software level can be a part of a mathematical graph or a union of disjoints parts of mathematical graphs.

As a conclusion, the graph manipulation API is not complex to develop, except probably the topology assertion mechanism (requirement #07) that can use non trivial graph isomorphisms algorithms.

5.1.4 About The Need For a Graph-Oriented Programming Language

Currently, there is no *graph-oriented programming language* available in the market.

[40] Some implementation may even trace what changed in order to commit only the modifications to the graph, in a single transaction.

Some open source projects attempted to introduce "graph-like" syntax. Gremlin for instance [21] considers the graph as a tree structure that can be navigated with the use of the "." (usually used for method access in Java). This, for us, is another way of generating huge coupling in the code, because the piece of code that will implement the traversal, will stick together a chain of nodes that, potentially, will not be linked this way tomorrow.

Having the capability to manipulate graphs natively inside the code may be interesting providing we do not loose our engineering objective: guaranteeing the long run maintenance and evolutions of enterprise software through the extreme limitation of unnecessary couplings. As we will expose in section 6 and followings, we will see that we are not blocked today by the absence of a graph-oriented programming language. Being in a functional environment or an object-oriented environment, we have all the required tools and technologies to enter the world of graph-oriented programming.

5.2 Software Structures Nearer From Business Concepts

In enterprise software, in the case of object-oriented programming, the question of manipulating business concepts inside the code is a central question of the industry for decades.

5.2.1 The Object-Oriented Business Layer

Most software engineers have realized for a long time that it was useful to have an *business layer* that represented as accurately as possible the business concepts [8].

Most of the time, this layer could not be strictly serialized inside a database. Engineering teams were facing the traditional questions of:

1. Database denormalization [34], i.e. the right way to adapt code and tables all constraints considered (prioritized constraints);
2. Adapting the business code through an *database adaptation layer* between business objects and relational tables, also called *data object layer*.

We can see, in the industry, several approaches to this issue:

1. The use of an Object-Relational Mapper (ORM) and manually coded adapters that take business objects to input them inside the data objects of the ORM;
2. The use of the ORM data objects *as* a business object layer [22];
3. The manual adaptation of the business objects and the tables through a custom data object layer that is enabling more fine use of SQL queries.

In the first and third cases, there is quite often a lot of adapting code between business objects and data objects. This code enables the software to be "not so coupled", because with this code, it is possible to decorrelate, in the development process,

changes in the code from changes inside the database[41]. In the second case, this is not possible: all must evolve at once.

Adapters are one of the mandatory consequences of object-oriented programming on software architecture. They can be used as a best practice but they are very heavy in the long run, and the service they provide (enabling partial changes inside the software) is costing a lot of money.

5.2.2 No Adapters in Graph-Oriented Programming

We saw in section 4.3.4 that graph-oriented programming proposed one single way of representing semantics throughout the various phases of traditional software development: functional analysis (semantic analysis), design and database.

In this section, we can say that our implementation guidelines of node and relationship types are completely bijective between the modeling perspective on one hand and the database on the other. Indeed, each node and relationship type can be created as respectively a node and relationship schema inside the graph database[42].

That implies that a lot of mappings and adapters that projects used in an object-oriented programming context, are no longer required:

a) Methodology mappings:

   i. The mapping between business functional analysis and object-oriented design is not required anymore;

   ii. The mapping between the design model and the code is not required anymore[43];

b) Adapters:

   i. The mapping between the business objects and the data objects is no longer necessary;

   ii. The mapping between data objects and the database can be managed quite efficiently.

5.2.3 About Object-Graph Mapping

The natural conclusion of what we just said is that the use of an object-graph mapper (OGM) can solve all issues in graph-oriented programming. We will temper this conclusion by the following comments.

It depends on what functionality an OGM provides and how it can enable the application code to reduce couplings. If we consider the graph library we briefly specified, the requirement #07 about graph persistence makes it an OGM. However, the storage capability is one feature amongst others. Our graph library is more a set of tools to manipulate graphs than a dedicated set of tools to persist graphs.

In particular, our graph library enables us *not to program in an object-oriented programming way*, whereas it is very easy to write an OGM that is just persisting objects in the traditional object-oriented design paradigm inside a graph database. In that case, the OGM is just changing the backend database but pushes for the continuation of the object-oriented programming paradigm and its associated troubles that we covered extensively in section 2.

As products are changing quickly, we just want to warn the reader about the various OGM products that are available and that will be in the coming years. Above all, those products must enable the graph-oriented programming paradigm that we will detail in section 6 and followings.

5.3 Treatments in Graph-Oriented Programming

To be able to have a complete view on the graph-oriented programming paradigm, we have to explain how the treatments are coded. As we saw in section 2, in object-oriented programming, the methods of the objects are melting a navigational knowledge (topology) and knowledge of methods of aggregated objects.

In graph-oriented programming, we have another way of "placing" the code that we do in object-oriented programming. Since the beginning of professional computer programming, code placement has always been a central focus point. We will introduce the notion of *graph transformation* to complete the exposition of the graph-oriented programming paradigm.

6 Graph Transformations

6.1 Introduction

The notions of graph transformation, graph grammars, graph rewriting, are widely studied from the 70's in the domain of theoretical IT (we can cite [37, 35, 40, 41, 39, 43, 42, 36] amongst other references). All those works are related to directed or undirected graphs, with ou without labels and colors.

We did not find, in all this literature, a lot of usable stuff for our problem, except in AGG [38] that was inspiring. The use of attributed directed graphs in enterprise software seems often quite far from the graph transformation topics that are explored in the research area. However, the research domain being very large, we absolutely do not claim to have explored the complete bibliography on the subject.

In enterprise software using graph databases, we have, at the same time, simpler and more complex problems that the problems that are generally dealt with in theoretical IT.

Actually the notion of graph transformation that we will introduce is the following: *a way to implement functional code, or business rules, that do not generate technical debt*.

Thus, we do not want to use more mathematical definition of

[41] In a web service based platform, we have the same concern between an evolution of the web service signature and the business layer, when it exists.

[42] When the database supports it, like OrientDB.

[43] There were massive works around the code generation from UML to C++ or Java and the parsing of manually written code to synchronize back UML models from code. Those works were not really convincing except in some particular cases where most of the code could be generated, which is not the case for enterprise software. The industry using UML seems to have converged to a rational use of UML (when they use it) in order to provide code maps and descriptions of tricky designs, instead of targeting the both ways synchronization of code and model.

graph transformation[44]. For us, graph transformation must remain an intuitive concept, as modification queries can modify data inside the database. For sure, as there are no more tables in a graph database, any query can be seen as a graph transformation (a select query being the `Identity` graph transformation).

We do not intend neither to provide *the* graph-transformation software, a software that would create an environment for the use of graph transformation in various domains or for graph rewriting (like PROGRES or AGG [38]). We intend to explain *how to build many graph-oriented software*, because they enable to get rid of coupling, and open the door to long term evolutions.

On top of that, our focus will be quite *microscopic* in the sense where we will transform localized areas of the graph, but we will not frequently get interested about *the full graph* as it is stored inside the database.

We will have also some specific problems that do not seem to be addressed by the IT research on graphs:

a) We have to deal with 3 levels of graphs: a design model that is a graph (graph of modeling artifacts like node and relationship types), the graphs of instances that we manipulate in the code, and the graph of nodes and relationships that are inside the database;

b) Our graphs are full of semantic issues related to the business areas covered by the enterprise software.

If the IT research area does not seem to bring many solutions to our problem, the IT industry is also not providing many things, OGMs being the only promoted tools that we can find in the area of graphs.

6.2 Modeling Graph Transformations

On top of our very large definition of graph transformation, we will characterize the graph transformation as follows: *A graph transformation is a function*[45] *(in the sense of a piece of code accessible through a functional interface) that takes in input a graph (or a subgraph) and gives in output a graph (or a subgraph).*

The output graph can be the same graph modified (destructive approach also called *with side effects*) or a new graph (no side effect approach[46]).

The fact that a graph transformation be a function is very important in the programming model that is, by many ways, *functional*.

6.2.1 Materializing Subgraphs

To be able to represent graph transformations graphically in a designer, we need to be able represent the input graph and the output graph. Instead of talking about graphs, we will talk about subgraphs because, inside the database, the complete graph will always be much larger than the software will be able to manipulate in a single transformation.

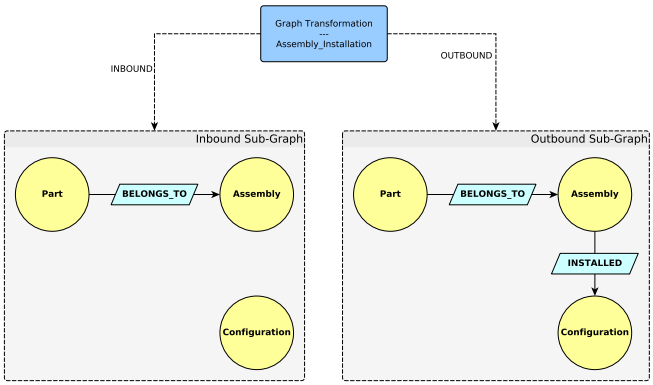

Figure 15: Representation of graph transformation

The subgraph notion exists in mathematics, but once again, our interpretation of the subgraph can be a bit different: a subgraph is for us any part of a graph, even incomplete parts (containing relationships pointing to nodes that are not in the graph, isolated nodes, isolated relationships, etc.). This must be managed by the graph library we described in section 5.1.3.

The subgraph will be materialized as a container artifact inside our design views. This container will be able to contain node and relationship types (see Fig. 15). The relation between node and relationship types and the subgraph will use the aggregation symbol of UML (see Fig. 16).

The link between the graph transformation and the subgraph will be a dashed arrow of one of two types: `INBOUND` and `OUTBOUND`.

The use of subgraphs in modeling enables to perform very quickly impact studies on graph transformations, node and relationship types (which will be very useful in the maintenance and evolution process).

6.2.2 Representation of Graph Transformations

Graphically, we chose to represent the graph transformation by a rounded blue rectangle (see Fig. 15). It is connected through dashed typed arrows to subgraph containers (through the `INBOUND` and `OUTBOUND` indicators).

It can be interesting to note that, in Fig. 15, we have several times the same modeling artifact instance. For sure, it is a view trick and the model contains just one `Part` node type.

The Fig. 16 shows a diagram that is equivalent (bijective) to Fig. 15 but that only represents entities once: it is more complex to understand the nature of the graph transformation with this representation.

We can note that the vision of Fig. 16 is much nearer from the real model structure. With graph model exploration, we will be able to perform impact study on the code, at the design level.

This is not so easy in UML where there is no way to externalize methods from objects. For instance, if a class `A` disappears from a UML model, there is no easy way in UML to clearly identify what are the dependencies, i.e. the classes that call

[44] Even if the complete set of business rules for an enterprise software could probably be considered as a graph grammar, we will not enter this debate.

[45] This term has to be taken in its conceptual sense. In an object-oriented environment, the function will probably be a method of some object.

[46] This approach is the standard functional approach.

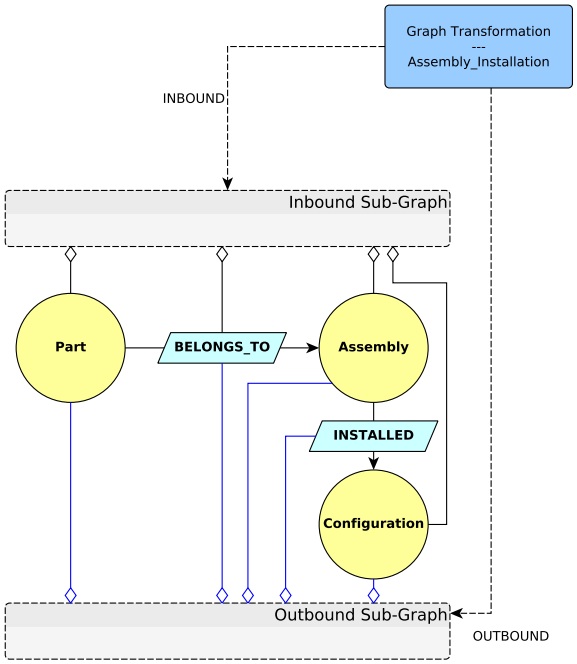

Figure 16: Subgraphs as containers

the methods of A. Some of those methods will probably appear in sequence diagrams but those methods are not complete artifacts: they are labels of artifacts.

With the graph-oriented modeling notation of graph transformation and subgraph, we can have a much more expressive power in modeling. Graph transformation are *first class citizens* from the modeling phase.

6.3 The Graph Transformation Code Structure

We are going to propose some rules to write code that will not generate technical debt and that will limit coupling to its most reduced expression.

Here is what we propose as a canvas for graph transformation structure. The graph transformation must take in charge three problems:

1. Check the input subgraph topology to determine graph transformation applicability;
2. Transform the inbound subgraph into the outbound subgraph;
3. Analyze and solve rewiring issues at the limit of the subgraph.

6.3.1 The Topology Check Step

The graph transformation must primarily check the input subgraph topology to see of the transformation is applicable.

We can find the notion of *applicability* in the enterprise software for industry. In that area, it is common to have programs manipulating complex data structures (that are indeed graphs, even if they are most often not manipulated as graphs). Those programs must determine if the business rules apply or not to the *configuration of data*.

We can note that this requirement is quite near from the design-by-contract paradigm [19].

By checking the topology of the graph, the graph transformation determines what nodes and relationships it expects to find, to be applicable. In case the topology is not compliant with the expected topology, the graph transformation will declare itself as `*NOT_APPLICABLE*`[47].

This check of topology must:

1. Be strictly limited to the required nodes and relationships;
2. Not presume about other relationships that may exist for certain nodes and that are not relevant in the context of the graph transformation;
3. Be expressed in a graph-oriented topology language enabling `assert`-like clauses.

For the graph topology check, we can imagine a DSL[48] [7] working directly in the graph library and checking topology through graph isomorphisms techniques [45].

6.3.2 The Graph Transformation Step

The real graph transformation will take place after the topology check. It can manipulate the subgraph or create a new subgraph.

There are various strategies in transforming graphs and the literature is quite big on the subject. In the case of enterprise software, graph transformations can be seen as a consistent set of queries that perform a meaningful semantic transformation relatively to the considered business domain.

6.3.3 The Rewiring Step

The challenge of the third step is to ensure that only relevant nodes and relationships were *seen* by the graph transformation. When the resulting subgraph is a part of a bigger graph, the modification must be done locally without damaging, by mistake, the relationships that are present but not relevant in the context of the graph transformation that is being executed.

This is a crucial point. It means that it is as if the graph transformation was acting on a *graph view* and was *blind* to whatever information that is not relevant to it. For instance, in Fig. 17, `h` is not modifying the inbound or outbound relationships of `N`. If `h` is correctly coded, it should not see any of those gray relationships and nodes.

This is the guarantee of the infinite and easy evolution of the software: provided the graph transformation does not assume more than it *strictly needs to know*, and provided the graph transformation may not apply in case of topology mismatch, we have the foundations of an ever-evolving system with no technical debt.

[47] Depending on the programming language, a return code or an exception can be used.

[48] Domain Specific Language.

6.3.4 The Rule of Evolutivity of Graph Transformations

We insist on the fact that graph transformations, in order to be evolutive, must never know or look at more relationships than it is required by the topology check. The change in the subgraph should always be local to the logical view defined by the graph graph transformation topology conditions.

This rule is crucial and not respecting it will generate bugs and unnecessary couplings.

6.3.5 Coupling in Graph Transformation

Considering the graph transformation structure, coupling exist through the knowledge of the subgraph. Indeed the coupling is (and must be) expressed in the graph topology checks.

We can note that:

1. This coupling is the minimal coupling possible considering the encoded business rule;
2. The knowledge of a topology condition is encapsulated with the rule itself (the graph transformation as a function checks its validity domain and applies only if it is applicable);
3. The graph transformation embeds transformation code that is not located *inside* nodes nor relationships (like in object-oriented programming) but in external functions;
4. Like in rule-based programming, the graph transformation is unique and can be traced with a unique ID[49].

The graph transformation can be seen as the *minimal coupling unit* in an absolute way: it only sees the view generated by the topology conditions acts on it without assuming more things about the "invisible" parts of the graph.

6.3.6 Philosophical Note

Philosophically, we can note that the graph transformation can be seen as *a kind of method of a graph*, the graph being *a complex object*.

Actually, in a lot of cases, the object-oriented approach is not sufficient or fully adapted to represent properly (complex) business concepts, whereas graphs are. In terms of treatments, the graph transformations are attached to the graph as the methods are attached to the object (through the topology checks).

Thus, graph-oriented programming can be interpreted as a natural evolution, or a generalization, of object-oriented programming: objects became (sub)graphs and methods became (sub)graph transformations.

This analogy can help us explain the real disappointment of the IT market concerning the object databases. We saw, from the beginning of this article, the limitations of the OOP/RDBMS technologies in terms of coupling generation. If the object-oriented programming paradigm was efficient in itself, the object databases would have been a real progress in enterprise software. The fact is object databases caused a lot of troubles that were not caused by relational databases, especially by forwarding the object-oriented coupling inside the storage itself (while object-oriented programming with adapters to the relational database can decorrelate the evolutions of the code from the evolutions of the database).

In our opinion, object-oriented design is not fully adapted for enterprise software because it generates, in itself, too many couplings.

Nowadays, a lot of initiatives intend to introduce into the IT industry new languages, especially functional languages (like Haskell[50] or Closure) or to push script languages that do not follow the strict object-oriented paradigm (like JavaScript, Ruby or Python). Those initiatives can be seen as attempts to program differently because of the limitations of the object-oriented programming approach.

We think most objectives that the inventors of the object-oriented programming paradigm tried to achieve seem to be achievable now with graph-oriented programming. If we consider that objects can be structured as subgraphs and methods are becoming graph transformations, we are quite near from the object-oriented programming spirit while solving most of the problems of this programming paradigm.

We can also say that the energy spent for decades to promote a programming paradigm versus the other (structured programming, functional programming, object-oriented programming) must be envisaged through the real benefits the programming paradigm brings. Graph-oriented programming is bringing the tools to soften software on the long run (notion of *soft-software*) by reducing coupling to its most minimal expression like we will see in the rest of this article. To our opinion, it is one of the best hope to change the way software is done and to prevent future generations to manage the couplings that we created inside the software our generation developed[51].

6.4 Composing Graph Transformations

6.4.1 Bottom-Up Programming

In current enterprise software, many programs are big and are taking in charge a lot of cases in one single piece of code. This code embeds various knowledge of data topology.

In graph-oriented programming, graph transformations are linked to their topology conditions, which means that, contrary to classical programming models, there should not be, inside a graph transformation code, business rules that apply to various topologies. In other words, graph transformations can be quite small, attached to a particular topology pattern. This way of proceeding enables bottom-up programming [13] on top of other programming styles.

[49] That is a requirement of some administration or military enterprise software to be able to put in a log the exact sequence of business rules execution. By rule-based programming, we mean a software built with some system of rule isolation (for instance, we are not referring to expert systems that can learn, we are focusing on predictable systems).

[50] Haskell is not a new language but its adoption by the industry seems quite new.

[51] A large number of companies are currently slowed down in their evolutions by the complexity of evolutions of their central system software or client-server (web) applications written in the 2000's.

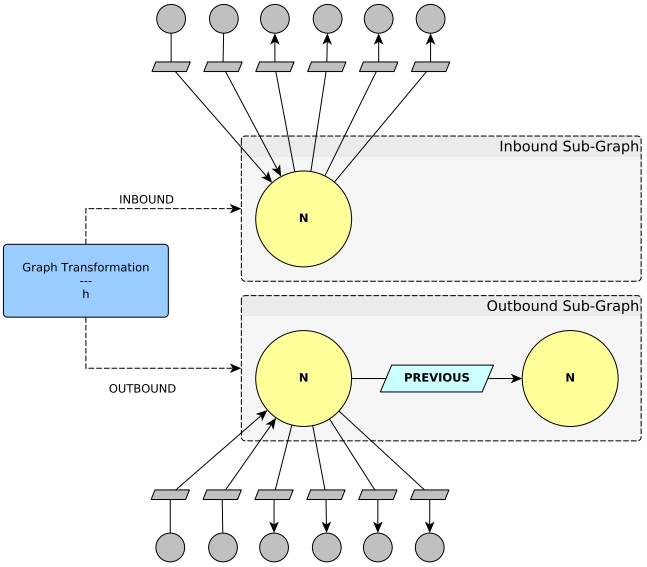

Figure 17: Rewiring preserves unknown relationships

Composition of graph transformations are eased by their uniform interface: subgraph in inbound and subgraph in outbound. The fact that they protect themselves from non applicability enables their easy composition even if, functionally, most composition will not make sense.

As a matter of facts, graph transformations appear as the most reduced *compose-able units of coupling* in the graph-oriented design software. They are the main kind of enterprise software *building blocks*.

6.4.2 Composition and Reusability

For sure, composition makes reusability possible, like we can see in the sample of Fig. 17.

We take the example of a function `h` that is doing timelining of nodes. The pseudo code of `h` is provided in table 5.

Note that `h` is, at the same time:

a) *With side effect*: because the subgraph `g` is modified,

b) *Without side effect*: because all the nodes and relationships of the initial graph stay untouched; The graph is *augmented* (at the node `n` level).

This point is very important because we can see that the old paradigms (including the functional programming one) are not sufficient to describe the properties of graph transformations. We will see soon that we can say the same about *intrusiveness* (see section 7.2.6).

By spawning a node "in the past", `h` preserves the inbound and outbound relationships of `n` which would not have been the case if the spawning had taken place "in the future". If `n'` would have been chosen to be the new root node[52], `n'` would have required the rewiring of all `n` relationships. `h` would have known the exhaustive list of `n` relationships, and so would have violated the rule of evolutivity of graph transformations.

[52] We explain in section 7.5.1 the *root node* concept.

Hypothesis
Let clone be the clone function. Let n be a node of node type N. Let PREVIOUS be a relationship type. Let createG be the function to create a graph. Let g = createG(n). We call g = h(g).
h is a function cloning the root node of a graph h: $G \rightarrow G$ G being the set of the database subgraphs
if g = null throw *NULL_GRAPH* if g.count() <> 1   throw *NOT_APPLICABLE* n = g.getRootNode() n' = clone(n) r = new(PREVIOUS,n,n') g.add(n',r) return g

Table 5: Pseudo code for the h graph transformation

This design choice solves the rewiring problem that we saw in section 6.3.3.

The `h` function can be reused by all graph transformations taking only one node. Let `f` be a transformation of this kind, `f ∘ h` is a correct graph transformation: it will spawn a clone of a business node in the past before taking it as root node for the `f` treatment.

6.5 Graph Transformation Evolution Rules

In this section, we will focus on the evolution of a business rule implemented as a graph transformation.

6.5.1 Proposed Evolution Rules

During the maintenance and evolution phase of the software, many "triggers" can imply an evolution of this business rule, though not all triggers will imply the same strategy in graph-oriented programming.

We propose the following *evolution rules*:

1. *If the topological applicability conditions change, then the graph transformation should be forked*;
   i. In other terms, an evolution of topological conditions creates a new rule that cannot be considered as an "evolution" of the previous rule;
   ii. Once the modification is done, the system contains two active rules (and two active graph transformation) with two different applicability conditions (probably on separate sets of data);
2. *If the topological conditions do not change, it depends on data time characteristics*:
   1. If all data in the database are submitted to the new rule (and none to the previous version), the rule can be modified ;

2. If some data still obey the previous version of the rule (generally "old" data), the rule should probably be forked and integrated into a timeline-based design; time-oriented coupling generation in the code can be an argument to choose one or the other solution.

Note that a change in data structure (node and relationship typed members) *is* a topological change[53].

6.5.2 Adding a New Attribute

The Standard Case

In the relational database case, a change in attributes often translates into a new column inside the database, which is generating an important change in the code. In graph-oriented programming, there is a choice to make, most often driven by the number of business rules that will be impacted by this new attribute.

In the relational case, the new attribute `a` will cause the table `T` to define a value for all rows, even for the rows that are related to functionally unchangeable data in the past. This process is equivalent to a (small) data migration (all data stored in `T` version `n` will be migrated to `T` version `n+1`).

In graph-oriented programming, we can do the same than in relational database. The problem is that it will require (as in the relational case) a potential retesting of all business rules accessing node type `T`. For sure, if the application contains no query of the kind `select * from`, the risk of facing a problem is low in both the relational and the graph case.

Alternate Design 1

The fact is, in graph-oriented programming, we can study an alternate approach.

If `a` has an important impact on existing business rules, related to new data created after a certain date, updating `T` can be questioned. In the relational case, that would mean including an unchangeable default value for `a` at migration time (for 100% of past data) and having meaningful values for `a` for new data.

It is complicated, in that case, to define a general evolution rule.

In graph-oriented programming, we can imagine a type $T' = T \cup \{a\}$, `T'` being an extension of `T` (node types are, in the implementation we imagine, classes), transformed by new graph transformations (`g1`, `g2` and `g3`) applicable to `T'` only (like shown in Fig. 18).

`g1` composes `h1` and casts `T'` in `T` (and the same for `g2`). `g3` is the replacement rule for `h3` and uses the new field `a`. In a certain way, this design is defining a new "interface" over `T`, the old one being [`f1`,`f2f3`] and the new one [`g1`,`g2g3`].

This design has a disadvantage: by construction, `f3` is applicable to `T'` whereas it should not be. This can be rather problematic because `f3` is not really "safe", despite its topology control.

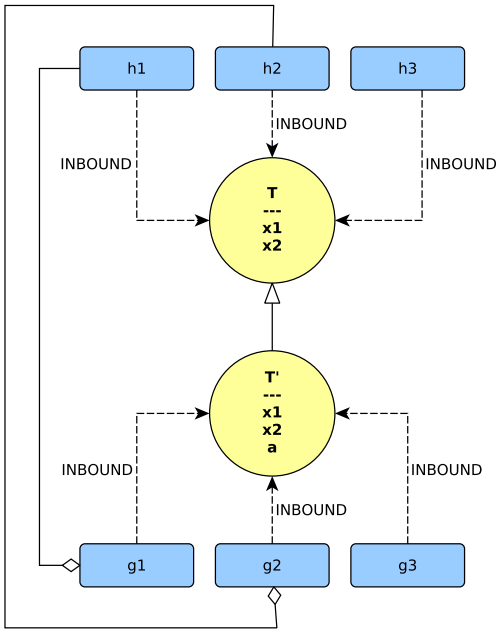

Figure 18: Field evolution, design 1

Alternate Design 2

A way to solve this issue is to declare that `T'` does not extend `T`, but an instance of `T` can be generated from `T'`: $k(T') = T' \setminus \{a\}$. `g1` and `g2` compose `k` on top of respectively composing `f1` and `f2` (see Fig. 19).

`f1`, `f2` and `f3` are only working on node type `T` and cannot take `T'` anymore.

This design is relevant in the case where `f1` and `f2` can work with a copy of `T'` so in the case where relationships allow it.

Alternate Design 3

Another way to design would be to consider that the application is timelined (see Fig. 20). Depending on a certain date (`Period`), the graph transformations `m1` and `m3`[54] are a melting between *façades* and *factories*[55] that can manage both `T` and `T'` with the proper transformation, depending on the time.

`m1` is in charge of calling `f1` with `T` or `k(T')`, and `m3` is in charge of calling `f3` with `T` and `g3` with `T'`. Both `m1` and `m3` have a temporal knowledge of the evolutions of business rules and the control of `Period` is belonging to the business rule part of `m1` and `m3`.

Do Not Miss The Initial Objective

The design of such cases should not miss the original objective that can only be determined from its business semantic content.

For instance, in the previous sample, if the change symbolized by the new attribute `a` is impacting heavily the software,

[53] At the time this article is written, graph databases are often very soft in attribute management and various objects can have the same type while having different sets of attributes. This is due to the Big Data requirements to manage partially structured data. When graph databases will propose strict schemas for node and relationship types, it will become obvious that two nodes with different typed attributes cannot be of the same node type.

[54] We omitted, in Fig. 20, the case of `f2` and `g2` for clarity and because it is the same case as for `f1` and `g1`.

[55] We could also imagine that `m1` returns the proper graph transformation depending on the node type. That design is perfectly valid but `m1` would not be a graph transformation anymore, which may be relevant in that case (see section 7.)

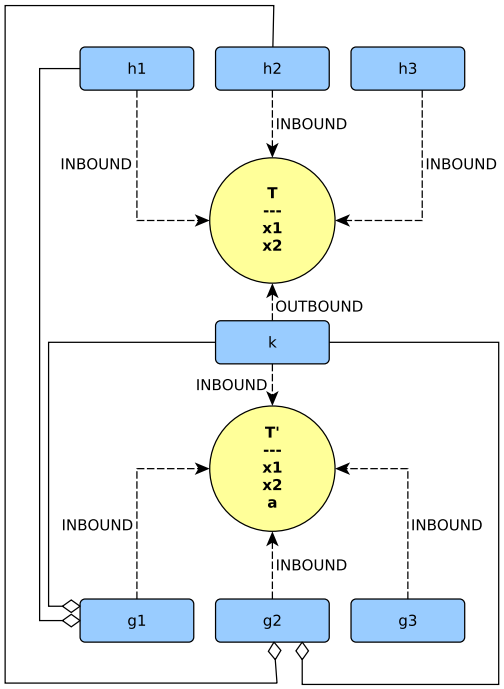

Figure 19: Field evolution, design 2

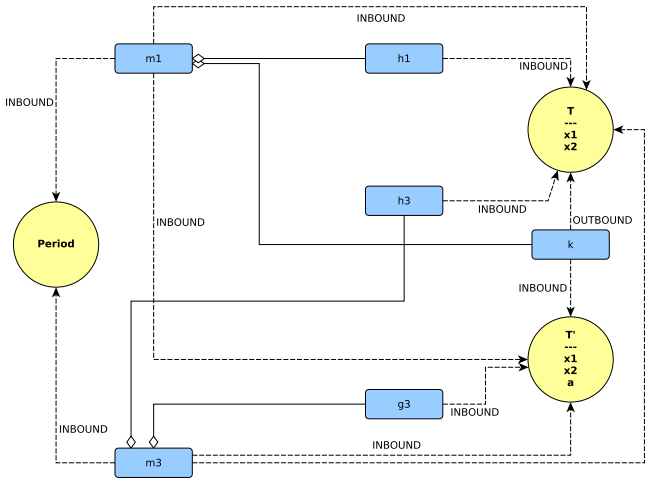

Figure 20: Field evolution, design 3

then, it can be useful to put previous structures and previous graph transformations in the past. In other words, design 3 could be realized with the intention of using directly `g3` in the future (and not `m3`). `m3` would be a graph transformation invoked in some past exploration part of the application, whereas the present application would focus on `T'` and `g3`.

If the addition of a field is just more data to the node type `T`, and a default value can be easily chosen for all `T` instances, and no graph transformation is querying the exhaustive list of attributes, then we can make `T` evolve.

Graph-oriented programming, in that case, only brings new possibilities. As we saw, even if fundamentally design 2 and 3 are simple, it seems mandatory to have a good modeling tool to be able to capture those design choices, especially in big software.

In current graph databases, it is often possible to have `T' = T` even if the list of attributes are not the same for `T` and `T'`. We do not push for the use of those features in enterprise software, because they may be misleading and source of code ambiguity.

6.5.3 Graph Transformations Are Not The Full Code

It is worth mentioning that graph transformations are not the full code. Within an enterprise software, there will be many pieces of code that will not be graph transformations. Graph transformations should be used for business rules around business data.

6.6 Timelined Enterprise Software

Graph-oriented programming enables to build a new generation of applications, applications that manage differently the time, what can be called *timelined enterprise software*. Depending on the various events that occur during the maintenance and evolution phase, there are ways to timeline properly business rules and business data depending on the business semantics.

This is particularly adapted for applications that manage time-sensitive business rules or regulations.

For instance, companies in the context of tax declaration, up to a certain date, declared 45 figures corresponding to well established rules. From the date $T0$, the law bounds companies to declare a 46^{th} figure. For the past declarations, it is of no use to migrate old data to the new structures, knowing that business rules applicable before $T0$ can still be applied for years in the context of controls. We will fork the business rule: new rule will manage new data structure and old rule will keep on managing old data structures. The calculation can still be done in the past in case of collection issue whereas other rules applies for the present.

The timelined applications will evolve while only paying the price of the new modifications. If a business rule is forked and is only applicable to new data, it does not put at stake what is already working inside the application and there is no use testing it on past (adapted) data.

Graph-oriented programming appears to be a simpler programming model than the programming models available today in the market. It also appears as a field of opportunities in terms of enterprise software.

6.7 Philosophical Note

Philosophically, the application is created to evolve by graph transformation and by data timelining. Social applications are already timeline-oriented so it will be very natural for the users to work on timeline-oriented enterprise software.

As the data and the code are concerned, we enter the world of *inflation*: if the productivity is constant with those applications and there is no technical debt effect, the application and the database will grow forever.

The real amount of code that will be maintained will probably not change much with time due to the design of timelined applications. Because on the present data, present rules will be used. In a certain way, the legacy code will keep on running on past data but will not be a weight for present applicable rules.

We are entering into the era of "inflationist enterprise software". Designers will, for sure, be of crucial importance because, in a growing world, navigators need maps to find their way home.

7 Graph-Oriented Programming Software Architecture

The graph-oriented programming paradigm have many positive consequences on software engineering. In this section, we will consider several aspects of those advantages that are more or less linked to software architecture concerns:

a) Utilities,

b) Domain-based extensions,

c) GUI considerations,

d) Time management and some other patterns.

The graph-oriented programming enables to consider differently reusable treatments and components, especially components that will offer some sort of persistence. The fact is, with the proper software architecture, we believe it is quite easy to implement those components once and have them enrich all enterprise software.

We will introduced the need for a controller inside graph-oriented programming enterprise applications, and we will detail how this controller could take in charge the "glueing" between reusable components and business code.

Concerning patterns, we will list some of them but we are sure that the literature will come soon with numerous other ones. We will talk about patterns when there is not, to our point of views, an easy way to create reusable components. Patterns in graph-oriented programming, like patterns in object-oriented programming, are answering to some generic design issues.

7.1 Super-type and Controller

In enterprise software (in an object-oriented world), classes are generally representing business concepts and can be instantiated in typed instances. Most of the time, they belong to a hierarchy of classes. We saw in part 4.1.5 that we did not want to use the specialization of business classes.

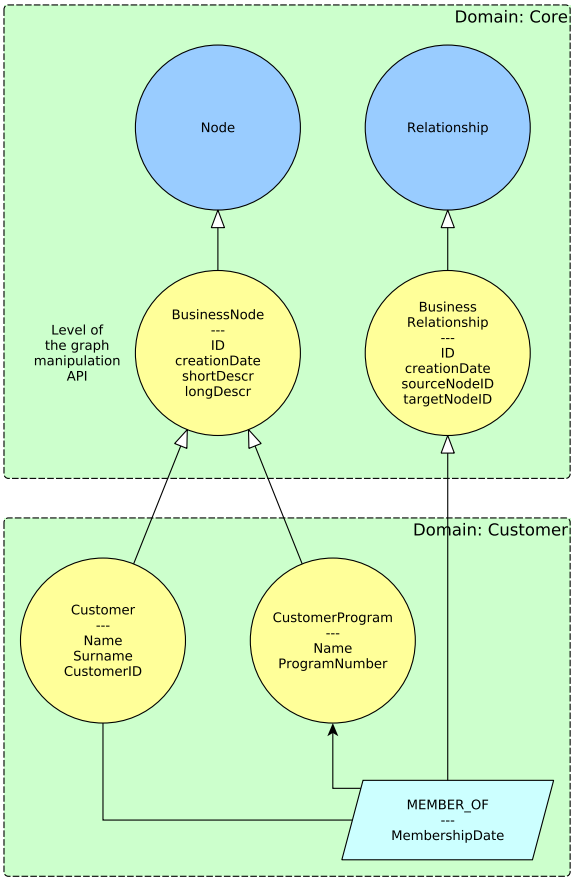

Figure 21: Strong typing of business node types and relationships

However, in this section, we plead for having a root type for business node types and a root type for business relationship types (respectively `BusinessNode` and `BusinessRelationship` in Fig. 21).

Having a root type for node and relationship types in the context of an enterprise application eases the implementation[56], for instance:

a) The ID management can be uniform;

b) The graph library will manipulate easily cast-able instances;

c) Some reusable treatments may be attached to the manipulation of casted nodes and relationships by `BusinessNode` and `BusinessRelationship`.

Concerning the last point, the Fig. 21 shows that we can always display the `shortDescr` attribute of a node because its type is extending `BusinessNode`. As the `BusinessRelationship` contains `sourceNodeID` and `targetNodeID` attributes, we can imagine a complete

[56] Some databases such as OrientDB already implement this feature inside the database itself.

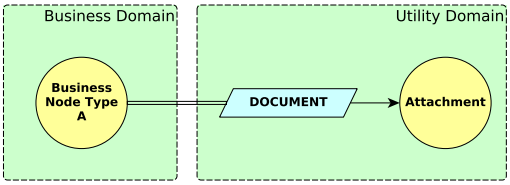

Figure 22: Decoration Utility: Attachment

generic graphical navigation system based on a kind of global *casted view of the database graph* (see section 7.5).

The coordination of those treatments and the articulation between reusable components and business code must be done through a kind of *controller*. This controller will be able to manipulate all business nodes as `BusinessNodes` and business relationships as `BusinessRelationships`. We will the controller notion several times in this section to show how the software architecture can benefit from graph-oriented programming.

7.2 A New Vision of Utilities

As we will see in this part, reusability is much easier in graph-oriented programming than in object-oriented programming.

In all those samples, we consider the presence of some controller being able to manage several domains of business objects "from the outside". This controller has a way to manipulate nodes and relationships of the business domains in an homogeneous way (see 7.1 and it manipulates some *utility domains* proposing reusable features.

We do not claim to be exhaustive in the utilities that could be created in a graph-oriented programming paradigm. The samples that we give are provided to demonstrate the basic mechanisms of graph-oriented programming and the power of the approach.

7.2.1 Decoration Utility

The first very simple case of features that can be bought by graph-oriented programming is the fact of attaching a *utility node type* to a business node.

In the sample of Fig. 22, we propose to consider an *attachment node* (type `Attachment`). It makes sense to attach documents, presentations, medias, etc., to many business nodes, to help characterize them. Moreover, the same document or media can be referenced by several nodes of various types.

In graph-oriented programming, in order to clarify the semantics, we will create a domain (see section 4.2 for the domain concept) with one node of type `Attachment` and one relationship `DOCUMENT`[57]. The `DOCUMENT` relationship is taking `*ANY_TYPE*` as source node type and is pointing to `Attachment`.

In this very simple example, the `Attachment` node will be a leaf node[58].

The important thing to notice is that *adding the utility domain will not modify any of the programs concerning the node type* `A`. The glue between the two domain (double line in Fig. 22 conforming to the convention we took in Fig. 11) will be managed by a controller that will be external to both domains (even if the controller will treat differently the utility domain and the business domain). In those conditions, nor the code representing node type `A`, nor the graph transformations acting on A will be touched by (or even "aware"of) the modification. For a strict "business code perspective", nothing has changed by adding the utility domain.

Due to this full decoupling, the decision of attaching attachments to any business node type can be done at any moment in the life cycle of the application. Moreover, it will cost almost nothing (because the utility itself is very easy to implement).

We can note that this feature can be implemented once, be packaged, and then be available afterwards for all enterprise software that will respect the controller convention (through some kind of plugin architecture).

If we compare that way of doing software to the way we used to do it in the OO/RDBMS world, we can discover the huge step ahead that we made. In object-oriented programming, we had first to modify all classes to point to a utility class (aggregation link to the `Attachment` class). Then, the tables had to be modified to include the persistence of the link (new column with document identifiers in all tables representing all business concepts pointing to documents). In case the document had to be deleted, the deletion had to be broadcated on all rows of all tables that were pointing to the specific document.

In graph-oriented programming, things are much simpler: many nodes can point to the same document and if the document is deleted, all relationships will disappear at once. If the controller manages this "utility" link from the "outside" (actually by using utility graph transformations managing the `*ANY_TYPE*` characteristic of the `DOCUMENT` relationship), the feature can be added:

a) For all business node types;

b) For almost no cost in the present;

c) With no technical debt generation nor coupling;

d) For all enterprise software.

Philosophically, this way of proceeding is a real game changer because this extension capability makes a lot of utilities free and potentially available in all enterprise software realized in graph-oriented programming approach.

By analogy with the object-oriented programming pattern, we can name this approach "decoration utility". We decorated a node type with some attributes that are grouped in a new autonomous and reusable node type and linked to the original node type through a utility relationship type.

[57] Note that we made a semantic choice. We could have name the relationship `MEDIA` or `IS_ATTACHED_TO` depending on our semantic intentions.

[58] This is for the purpose of the sample because, in a real enterprise software, attachments are generally grouped in more complex libraries.

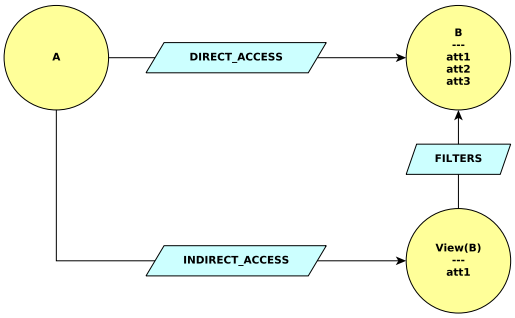

Figure 23: Basic View Concept

An important example of this design is the geolocation utility (see Fig. 10). The `Location` node type will belong to a utility domain that will propose services to find locations near from another location. Adding geolocation to an existing business domain will be easy, without coupling and free.

Considering the simplicity of implementation of those features, their integration and maintenance costs in any enterprise software should be *null* (!).

7.2.2 Views

As the decoration utility is somewhat "extending" a business node with new attributes grouped inside a new node type and linked through a relationship type (and forming one reusable node type and one relationship type), the view mechanism is working the other way round: A certain node type offers a façade to another node type hiding some information from it.

In Fig. 23, we create a view `View(B)` on the `B` node type. This view is accessible through a relationship type `INDIRECT_ACCESS` that offers another path than the `DIRECT_ACCESS` link. This can be a way to hide information from the caller `A`. The issue here is that the `View(B)` and the `B` node types have to be synchronized in some way.

Actually, this mechanism is very near from the "materialized view" concept that we find in relational databases, except that it can be done at the node level.

We can note that it is not a utility because it does not generate reusable types, but it is more a classical pattern.

7.2.3 Shortcut Utility

A variation of the attachment node can be called the *shortcut*. The concept is that, for certain reasons, the enterprise software user would like to link two nodes together (very often the reason is based on the easy navigation between two nodes which grouping is relevant in a certain user process). The framework will propose an utility relationship named `SHORTCUT` that will enable such an unplanned linking of nodes.

The Fig. 24 shows the various domains involved in the operation.

Like in the decoration utility, this way of proceeding has no structural impact on the code of any of the two business domains involved. The utility will be managed by the controller

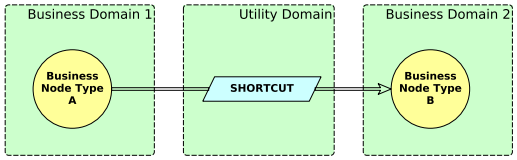

Figure 24: Shortcut Utility

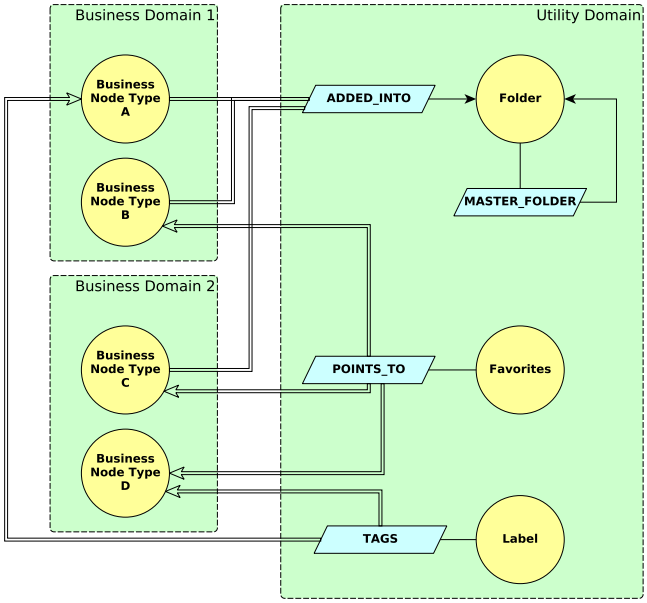

Figure 25: Container Utilities: Folders, Favorites, Labels

that will be able to "translate" this utility relationship into an *alternate navigation* capability.

This can be very powerful in the customization of the use of the enterprise software by users. That is a fact that enterprise software are generally used by many different users that are working under various processes. Instead of having to choose what process will be implemented and what processes cannot be implemented, enterprise software that will use graph-oriented programming will implement the main process and let users modify more or less extensively their user experience to fit their own processes.

We can note that the relationship `PREVIOUS` in Fig. 13 is also a sample of a particular kind of shortcut utility (restricted to chaining `JournalEntry` instances).

7.2.4 Container Utility

The *container* utility is the exact consequence of what we have just seen. Instead of linking together two business nodes, many business nodes will be grouped "inside" the same container.

Semantically, a lot of business classifications are containers even if they have different names and features:

- Folders may contain folders and business nodes,
- Favorites may be not-nestable and personal,

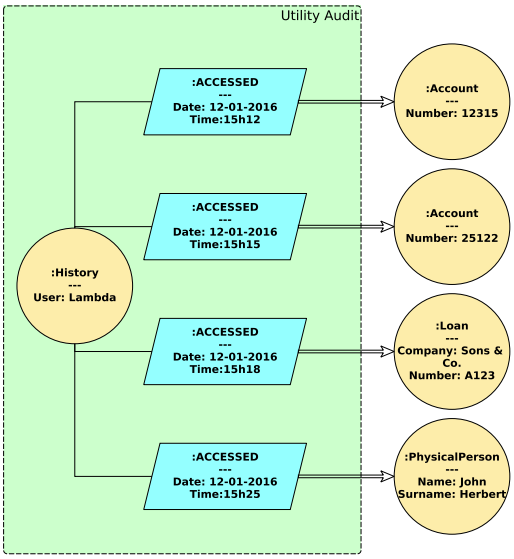

Figure 26: The History Facility

- Tags may be share-able between several users.

Note that we could also consider other container types "project", "organization unit", "catalog", "nomenclature", etc.

In Fig. 25, we see samples of containers in a utility domain. This domain could be extended with a lot of features (graph transformations) to manage properly the content of it. For instance, the `Folder` container type could be associated to access right management, multi-criteria classification, sharing rules, archiving rules, etc. It could also be coupled with a node type representing an attachment (like exposed in Fig. 22).

7.2.5 User History Utility

Let us consider another example: the user history (shown in Fig. 26).

The instance diagram shown in Fig. 26 is representing one way of recording the history of visited nodes by a user (even if this way may not be the most efficient way).

Formally, the mechanism used is very near from the container approach except that the relationships enable the sorting of information in another domain: the *audit* one. Once again, the audit domain could propose a large range of functionality that are not shown here.

7.2.6 Extension Without Redesign

The development of those utility domains is independent from the development of the business domains (like we exposed in the multi-domain approach in section 4.2). Moreover, once available, this utility domain can *enrich all enterprise software* without any overcost nor technical debt.

The fact of being able to extend an application without being intrusive inside the code is a bit of a revolution. Once consequence is also that, depending on the controller architecture, it is not necessary to migrate the data and to non-regress the full application. Literally, we could say, that the graph-oriented

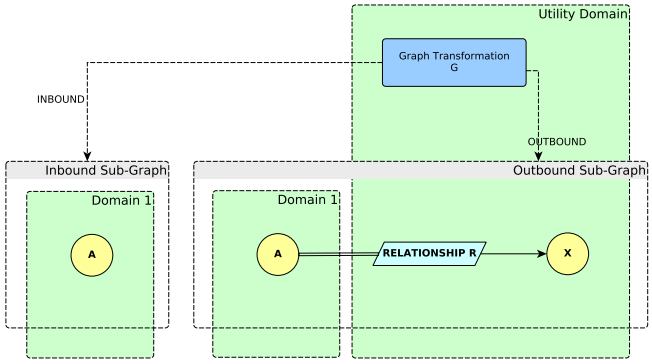

Figure 27: Gluing Domains Together

programming paradigm implies a capability of extending the enterprise software without extension, data migration and non-regression testing.

In the rest of the article, we will call that simply: *extension without redesign*.

We can note that those extensions without redesign are, at the same time, *intrusive* inside the core model (because they plug *into* the domain) and *not intrusive* because they have no impact on it.

The reason is located in the graph transformation structure: considering they do not know more than they should and do not assume something about the relationship they do not know, business nodes can be linked to utilities without any functional code to be impacted.

This is a very strange but very powerful feature of graph-oriented programming, and it gives a quite different look on reusability and extensibility.

7.3 Gluing Domains Together

As we saw in section 4.2, the domain notion is an abstract notion. However, it can be materialize at some point by a "package" notion containing node and relationship types and graph transformations.

As utilities are concerned, if utility services can be connected to a business domain (for instance the domain of `A` in Fig. 27), we need to examine how it can work in terms of graph transformations.

The Fig. 27 proposes a solution to this problem. The graph transformation `G` will link an instance of `A` to an instance of `X`. In order for the utility treatment to work without generating coupling, the graph transformation `G` must be able to manipulate `A` without knowing exactly its exact type[59]. `A` must be "casted" by a super-type.

Secondly, the caller of `G` must provide `G` with a casted `A`. Once again, we discover a need for a controller that would manage the integration of all parts together in a plugin approach, like shown in Fig. 28.

We can note that, in Fig. 28, we show a gluing done through a super-type that is `BusinessNode`. Indeed, we could

[59] That would mean in Java for instance without including a dependency to the package containing A.

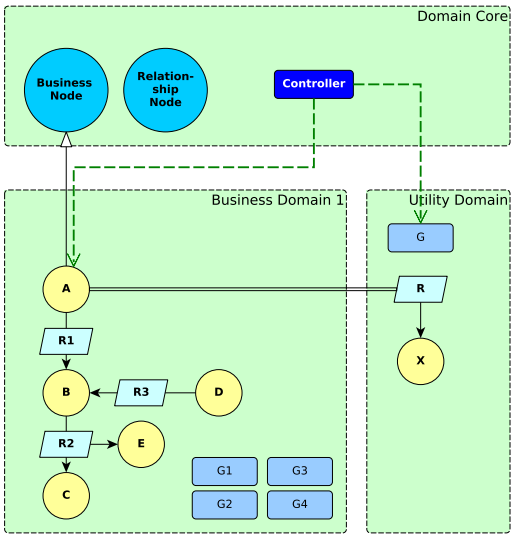

Figure 28: Gluing with Supertype

imagine the super-type to be much more generic and ony be `*Node_Type*`. But, we will see in the rest of the article that an intermediate type like `BusinessNode` has some advantages in terms of generic displays.

7.4 Domain-Based Extensions

One consequence of the multi-domain approach described in section 4.2 is that many business domains can be integrated inside the same application. For memory, we can associate together in a business domain:

a) Business node types,

b) Business relationship types,

c) Graph transformations (and their associated subgraphs).

Most enterprise software are covering several business domains. To identify those business domains, architecture languages such as Archimate[60] [17] are very useful.

Let us take an example taken from the aerospace maintenance. A common aerospace maintenance information system will cover (more of less completely) the functionality of the following domains: aircraft configuration, maintenance tasks, inventory, procurement, CRM[61], invoicing, shipments.

For sure, all those domains will propose inward treatments and business rules (graph transformations) and connectivity to other domains through inter-domain relationships (see also **??** for a comparison with mathematical concepts).

The proper management of domain gluing will determine if we generate coupling and technical debt or not. As we saw in section 7.3, gluing of a business domain and a utility domain can be done through a super-type of node types.

[60] Archimate is an enterprise architecture modeling language standardized by the Open Group.

[61] Customer Relatonship Management.

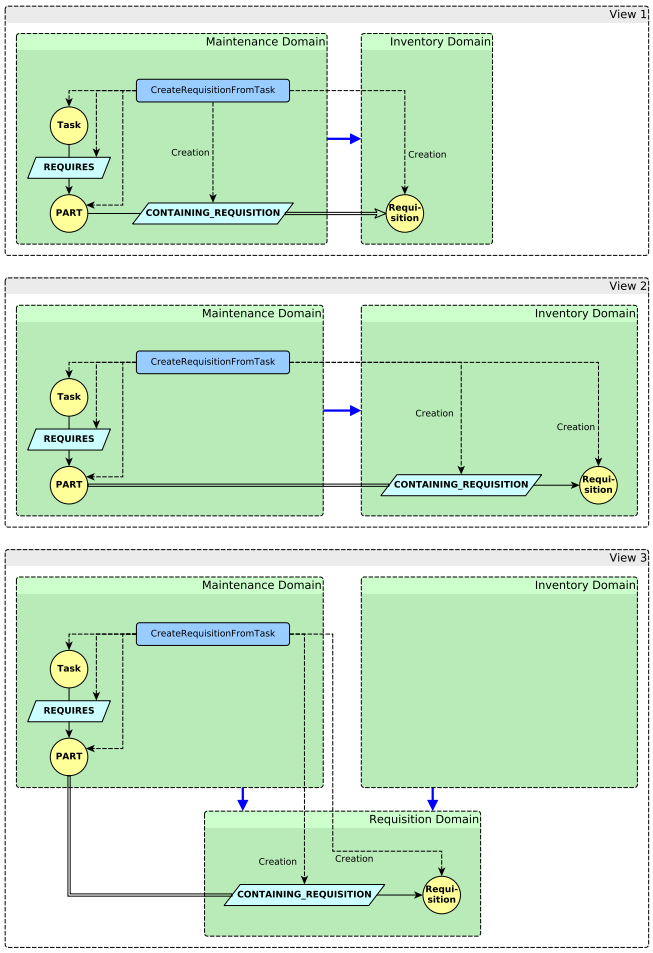

Figure 29: Three Solutions for Gluing Business Domains

The utility considers all nodes, whatever their real type, to be `BusinessNodes`.

When gluing two business domains, we need, at a certain point, for a graph transformation of domain 1 to know about one node type of domain 2. If we keep on with our previous example, we can take the requisition. A `Task` will reference `Parts` (in the maintenance domain) and those parts must be provisioned. They are requisitioned in the inventory, `Requisition` being a node type of the inventory domain.

Clearly, we have a functional dependency between the maintenance domain and the inventory domain: the maintenance domain "knows" a part of the inventory domain and the reciprocal is false. Even if we don't want to enter too much into the software architecture issues, we can propose two ways of solving this issue:

1. The direct gluing;
2. The indirect gluing through an "interface" domain.

7.4.1 Direct Gluing of Business Domains

The problem statement is simple: only a action in the maintenance domain can trigger the creation of a requisition (in the inventory domain). Consequently, there

will be a graph transformation of the maintenance domain (`CreateRequisitionFromTask`) that will "know" how to create the relationship `CONTAINING_REQUISITION` and the business node type `Requisition`.

The Fig. 29 is proposing in View 1 and View 2, two solutions of direct gluing, respectively with the relationship belonging to the source domain (maintenance), or the relationship belonging to the target domain (inventory). In both cases, the dependency between domains is shown in blue in the Fig. 29.

In that case, the business semantics is driving the coupling between the two domains. Graph-oriented programming cannot make this semantic dependency disappear.

This case is quite common in object-oriented design and it is one of the problems the software architecture solves: what packages should depend from one another. In graph-oriented programming, those software architecture topics remain when semantic dependencies cannot be avoided.

7.4.2 Indirect Gluing of Business Domains

There is however another solution to glue the two domains together and it is exposed in the View 3 of Fig. 29. The dependencies are transformed into a dependency from the maintenance domain to the requisition domain (which plays the role of an *interface* domain) and into a dependency between the inventory domain and the requisition domain.

In big enterprise software, those ways of interconnecting domains can be quite useful because the two domains maintenance and inventory may be sold separately (and so have the obligation to run independently).

The architecture proposed in the View 3 is not always generalizable. It depends on the level of interdependency between domains (see also a comment on that point in section **??**).

To deal with those software architecture problem, we plead for a larger usage of architecture languages such as Archimate. Those language enable multiple views of the same problem and a quite efficient management of dependencies.

7.5 GUI Considerations

7.5.1 Subgraph and Root Node

If we look at the way the object-oriented applications are built, enterprise software screens show us data. Those data are, most of the time, centered on a particular object and focus on some connected objects and/or some lists of connected objects.

In a graph world, strictly speaking, each screen is a view on a subgraph. Those subgraphs are quite particular because, if we keep the same approach than in object-oriented programming, they all have a root node (which is quite often the result of a previous query). By root node, we mean the main concept represented on the screen.

Any interaction that we are doing with the GUI can trigger a graph transformation on the subgraph represented on the screen (or a different subgraph).

When all root nodes are instances of `BusinessNode`, the root nodes can be manipulated by a generic navigation that can:

a) Manipulate individual root nodes (for instance to find their associated display);

b) Generically display lists of nodes, whatever their type, and through the display of `BusinessNode` attributes.

7.5.2 About Screen Navigation

In an object-oriented programming world, screen navigation is often hard coded inside the application, because there are structural relationships between entities both at the code level and at the database level (see section 2.1).

In a graph world, there are no more structural relationships on data, and with graph-oriented programming, there are no more unnecessary coupling inside the code. That means that any workflow defined at a certain moment is only representing a certain "path" between nodes, but that that path may evolve with time without any structural impact on the application.

Thus, we can see a real *opportunity* to build more dynamic GUI workflows.

7.5.3 Alternate Workflows

If we consider that any screen can be associated with a root node, we can define a screen workflow by a flow between root nodes. For sure, root nodes may not be connected directly like shown in Fig. 30.

The chain of root nodes is `WorkReport` then `Mechanic`, the `Task` node type in between being displayed in screen 1 as an element of the subgraph which root is `WorkReport`.

If we consider the `Task` node type, we can see in gray that it is connected to many other types that are not displayed in the main application graphical bean. Those nodes are both incoming and outgoing. All those connections could represent pertinent navigation options.

If all nodes and relationships are instances of respectively `BusinessNode` and `BusinessRelationship`, a framework can propose those alternate navigation options by displaying all the relationships that are not already displayed by the user graphical bean representing the root node (bottom zones in screen 1). Utility links can be displayed this way.

7.5.4 About Workflows

In certain constrained processes, there is no possibility for the user to interrupt a series of screens. The user is bound to realize things in a particular order, in an "ordered set of successive actions".

In OO/RDBMS applications, it is complicated to determine if the workflow of screen has a business meaning or of it is the image of object aggregations and table joints. In graph-oriented programming, as there are no more structural aggregations or joints, the navigation should be a real new experience. Software engineers will realize that most "commonly accepted" workflows are just the reflection of technical constraints. Real workflows should be rarer in graph-oriented programming than in object-oriented programming.

This approach is a real step forward in the GUI building compared to our current way of doing software engineering.

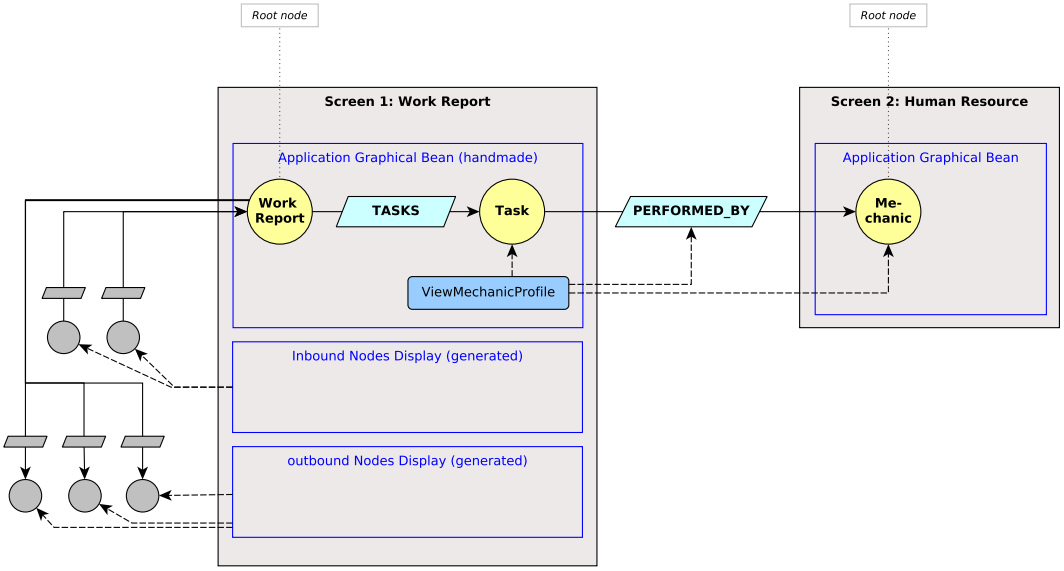

Figure 30: GUI Opportunities For Alternate Navigations

For a long time, software architects were bound to choose the *less worst compromise* in order to give satisfaction to the most "relevant" customer workflows. This problem is one pathology of ERP maintenance and evolutions: the ERP vendor would like to accommodate all relevant change requests coming from many various customers. But he cannot, because he is prisoner in the past software design choices.

With the graph-oriented programming paradigm, this syndrome can cease in two dimensions:

1. Relevant workflows will be implementable at any time whatever the state of the current software;

2. Some utilities such as the shortcut or one favor of containers will enable the customization of the user experience *by the user himself* (!).

Actually, some enterprise software workflows have to be mastered in order for the user to respect the various steps that were defined during the software specifications. For sure, even if graph-oriented programming enables to propose alternate navigations for free does not mean that it is always a good choice to let this option opened inside an enterprise software.

7.5.5 Another Way of Considering Training

Training, in enterprise software projects, is a huge part of the costs. Before (or behind) training, we often find change management topics, including process changes for some teams to adapt to the logic of the new software.

The generalization of utilities in the software (when it is relevant) can offer an alternative to extensive process change management and training. It can enable groups of users to use differently the same software. Many times in our project experience, we saw this need for a slightly different approach in workflows, for certain perfectly valid business constraints, that could not be taken into account in the project scope due to the fact that this use of software was considered as "at the margin".

With all those features around containers and alternate navigations, end users will be able to adapt their use of the software by themselves with fewer training or no training at all. If the application are well designed, there should be a lot of freedom for the user and a real improvement of the user experience. The company intentions will not be "train everyone to the same process" anymore but learn by doing inside the application.

Graph-oriented enterprise software will be much easier to discover and use, and much more natural to understand because it will respect the natural semantic (evolving) links between concepts.

7.6 Another Way of Managing Time in Software

As we saw in section 2, current enterprise software often consider one single code should manage every version of data structures ever used in the application, all those versions being migrated into the recent data schema.

We already saw in sections 6.5 and 6.6 samples of timeline designs.

The topic of timelining enterprise software is very big and should give birth to numerous patterns in the future, because the problem is at the same time a very interesting design problem and one of the crucial stakes of complex software creation. Graph-oriented programming is providing easiest ways to create timeline software than ever before. With graph transformation evolution rules, we can imagine software that will evolve and adjust to present data structures and business rules, letting in the past the old business rules and data.

Even in the case of time management inside the software, which appears to be the most complicated case, redesign of past constraints, data migration and global non regression testing should not be required anymore in a lot of cases.

7.7 Recommended Approach For New Graph-Oriented Programming Enterprise Software

For the application rewriting projects, we advise, in a first step, to reimplement the enterprise software with the graph-oriented programming paradigm while sticking to legacy workflows. For sure, alternate navigation mechanisms should be available as much as possible (see the discussion about alternate navigation in section 7.5.3).

Users will not be destabilized by the new software but they will benefit from new ways of working. After a while, we believe that a large proportion of users will have a custom use of the software.

Then, in a second step, the IT teams can propose new features that require new workflows (and imply new relationship types, see section 7.5.4). As the maintenance and evolution will not generate technical debt anymore, all evolutions can be added to the existing software following the rules of the graph-oriented programming paradigm.

By using graph-oriented programming, designers will not be bound anymore to choose what category of users will have their processes implemented inside the software. All departments of a company will be able to use the same software in very various ways, the limitation being always the business semantics.

8 Conclusions

We tried, in this article, to provide a first glance at the graph-oriented programing paradigm. We believe this programming paradigm is a game changer and enables to create, for the first time, in enterprise software, software that will be infinitely evolutive and that encapsulate coupling in disjoints units (graph transformations).

Many topics could be covered, such as:

1. The big advantages brought by the use of the graph-oriented programming paradigm in a SOA[62] backend application;
2. The easy and powerful extension of a graph-oriented programming enterprise software by Big Data techniques inside the same database and the many IT architecture advantages of such evolution.

The graph-oriented programming paradigm:

1. Is *simpler* that the object-oriented programming paradigm;
2. Can manage a *higher complexity* and timelined software;
3. Is *cheaper* than current technology;
4. Is *nearer* from the business concepts and the very semantics of business;
5. Is generating *no technical debt*;
6. Is fun;
7. Is filling the gap between business people and IT people;
8. Is enabling to build infinitely extensible applications with a constant function point price through the phase of maintenance and evolutions;
9. *Is requiring modeling*.

The graph-oriented programming paradigm seems to us as being the conclusion of the first software era, and the opening of a new one.

For decades, since the apparition of software, software engineers have searched for ways to reuse and to develop better. And all attempts converge to graph-oriented programming: functional programming, object-oriented programming, design by contract, rule-based programming.

Even if those abstract preoccupations come and go because of technology hypes, financial constraints and project time-to-market, several generations of software engineers have searched new ways of doing software.

Graph-oriented programming is very simple, it is near from all what exists and still, it is a revolution of the mind. With the graph-oriented programming paradigm, we believe enterprise software can be changed deeply.

Graph-oriented programmingwill change the maintenance and evolutions phases. Those phases will not require anymore redesign of past designed stuff, data migration and full non regression retesting. The maintenance costs will decrease which will enable:

a) A complete reallocation of resources in the IT departments;

b) A much better productivity of software maintenance and evolutions;

c) A easier customization of enterprise software to any specific need;

d) A much better agility in the enterprise software systems;

e) An extension of the reusability concepts that we know today because of the “extension without redesign” pattern.

Due to the graph-oriented programming paradigm, many aspects of the IT business could be impacted.

9 References

9.1 Software Books References

[1] D. Alur. *Core J2EE Patterns*. Prentice-Hall, 2^{nd} edition, 2003.

[2] J.-P. Banâtre, S. Jones, and D. Le Métayer. *Prospects for Functional Programming in Software Engineering*. Springer-Verlag, 1991.

[3] W. Brown. *Anti Patterns - Refactoring Software, Architectures And Projects In Crisis*. Wiley, 1998.

[62] Service-Oriented Architecture.

[4] F. Buschmann. *Pattern Oriented Software Architecture Volume 1 - A System of Patterns*. Wiley, 1996.

[5] J. Darlington. *Functional Programming and its Applications, An Advanced Course*. Cambridge University Press, 1^{st} edition, 1982.

[6] H.-E. Eriksson and M. Penker. *Business Modeling with UML: Business Patterns at Work*. Wiley, 2000.

[7] M. Fowler et al. *Domain Specific Languages*. Addison-Wesley, 2011.

[8] E. Evans. *Domain-Driven Design, Tackling Complexity at the Heart of Software*. Addison-Wesley, 2003.

[9] M. Fowler. *Analysis Patterns, Reusable Object Models*. Addison-Wesley, 1996.

[10] M. Fowler. *Refactoring, Improving The Design Of Existing Code*. Addison-Wesley, 1999.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements Of Reusable Object Oriented Software*. Addison-Wesley, 1994.

[12] H. Glaser, C. Hankin, and D. Till. *Principles of Functional Programming*. Prentice Hall, 1^{st} edition, 1984.

[13] P. Graham. *ANSI Common Lisp*. Prentice Hall, 1996.

[14] T. Halpin. *Object-Role Modeling Fundamentals: A Practical Guide to Data Modeling with ORM*. Technics Publications, 2015.

[15] T. Kowalski and L. Levy. *Rule-Based Programming*. Kluwer Academic Publishers, 1996.

[16] P. Kruchten. *The rational unified process: an introduction*. Addison-Wesley, 2^{nd} edition, 2004.

[17] M. Lankhorst. *Enterprise Architecture At Work - Modelling, Communication and Analysis*. Springer, 2^{nd} edition, 2009.

[18] B. MacLennan. *Functional Programming, Practice and Theory*. Addison-Wesley, 2^{nd} edition, 1990.

[19] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2^{nd} edition, 1997.

[20] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly, 2^{nd} edition, 2015.

9.2 Software Articles References

[21] Apache. Apache tinkerpop, 2016. See URL at apache.org.

[22] A. Bien. Real world java ee patterns - rethinking best practices, 2009. See URL at java.net.

[23] W. Cunningham. The wycash portfolio management system, 1992. See URL at c2.com.

[24] N. Brown et al. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010.

[25] M. Fowler. Technical debt, 2003. See URL at martin-fowler.com.

[26] J. Hugues. Why functional programming matters. 1990.

[27] D. Longstreet. *Function Points Analysis Training Course*, 2008. See URL at softwaremetrics.com.

[28] R. Martin. Java and c++ - a critical comparison. 1997. See URL at cleancoder.com.

[29] G. Nordstrom. *Metamodeling - Rapid Design and Evolution of Domain Specific Modeling Environments*. Thesis, Vanderbuilt University, May 1999.

[30] Oasis. ebxml message service specification v2-0, 2002. See URL at ebxml.org.

[31] Object-Management-Group. Object constraint language, 2014. See URL at omg.org.

[32] Open-Group. Dce 1.1, remote procedure call, 1997. See URL at opengroup.org.

[33] R. King R. Hull. Semantic database modeling: survey, applications, and research issues. 1997.

[34] G.-L. Sanders and S. Shin. Denormalization effects on performance of rdbms. In *Proceedings of the* 34^{th} *Annual Hawaii International Conference on System Sciences*. IEEE, 2001.

9.3 Graph Grammars References

[35] V. Claus, H. Ehrig, and G. Rozenberg. *Graph-grammars and their application to computer science and biology,* 1^{st} *International Workshop*. Springer, 1978.

[36] B. Courcelle and J. Engelfriet. *Graph Structure And Monadic Second-Order Logic*. Cambridge University Press, 2011.

[37] E. Ehrig, M. Pfender, and H. J. Schneider. Graph grammars, an algebraic approach. 1973.

[38] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation, Applications, Languages and Tools*, volume 2. World Scientific, 1999.

[39] H. Ehrig, H.J. Kreowski, and G. Rozenberg. *Graph-grammars and their application to computer science,* 4^{th} *International Workshop*. Springer, 1990.

[40] H. Ehrig, M. Nagl, and G. Rozenberg. *Graph-grammars and their application to computer science,* 2^{nd} *International Workshop*. Springer, 1982.

[41] H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfled. *Graph-grammars and their application to computer science, 3^{rd} International Workshop*. Springer, 1986.

[42] U. Prange H. Ehrig, K. Ehrig and G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer, 2010.

[43] G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation, Foundations*, volume 1. World Scientific, 1997.

9.4 Mathematics References

[44] R. Diestel. *Graph theory*. Springer-Verlag, 2000.

[45] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphism. 1979.

10 Appendix A: Mathematical Considerations

It is possible to interpret some of the features we mentioned in a slightly more mathematical way. Instead of talking about nodes, we will, in this section speak about vertices and instead of talking about relationships, we will speak about edges [44].

10.1 Concepts Taken From Graph Theory

10.1.1 Two Kinds of Graphs

In graph-oriented programming, we have two kinds of graphs:

1. The graph model,
2. The graph of data (inside the database).

Treating each of them with mathematically inspired concepts can be sometimes useful.

10.1.2 Not Connected Edges

In graph-oriented programming, we have to manipulate not connected edges. Those objects are not strictly speaking "complete" mathematical (or database) objects. For instance, an edge with a source vertice and no target (or the reverse) or an edge without any vertice attached. Those use cases are valid in graph-oriented programming, through the graph manipulation library. They represent intermediate state of the software.

10.1.3 Subgraphs

In mathematics, a lot of graph properties are taken in the context of a full (finite) graph.

In graph-oriented programming, we have graphs that can be very large, but programs will only work at the subgraph level. Mathematical graph properties can be interesting but our interpretation in IT will always be at the *subgraph* level.

10.1.4 Domains

If we look at the graph model, domains are containers linking many model vertices (node and relationship types, graph transformations). The domain can be considered as an *hyperedge* linking all vertices of the domain. The graph model can be seen an *hypergraph*.

Domains can also be seen as near from the concept of *graph component*.

10.1.5 Connectivity

The connectivity concept is very important in graph-oriented programming, being for utilities, domain connectivity or alternate navigation.

Pivot nodes can be seen as *cutvertices*. `Requisition` is a cutvertice in Fig. 29 view 3.

Connecting relationships are *briges*, for instance `DOCUMENT` in Fig. 22.

10.1.6 Paths

Most of the reusable components and patterns that we spoke about in the 7 section are creating new *paths* between vertices. For instance, geolocation or containers create new paths in data, as alternate navigation is exploiting the paths available in graph data.

The property of graph transformations to analyze topology to determine the applicability must be robust to the addition of new paths in all vertices in the inbound subgraph.

The creation of new paths is a fundamental property of graph databases: paths can be added between vertices at any time in the life cycle of the application. Considering the code is robust to path addition, the graph-oriented programming paradigm proposes the better evolutivity system ever.

10.1.7 Distance Between Vertices

The path notion enables to calculate the distance between two vertices. We can classify the reusable features and patterns from section 7 by distance ($Dist$).

Decoration Utility

Considering Fig. 22:

$$Dist(A, Document) = 1 \tag{1}$$

Views

Considering Fig. 23, with the view pattern:

$$Dist(A, B) = 2 \tag{2}$$

Shortcut

Considering Fig. 24:

$$Dist(A, B) = 1 \tag{3}$$

Container

Considering Fig. 25:

$$Dist(A, Folder) = 1 \tag{4}$$

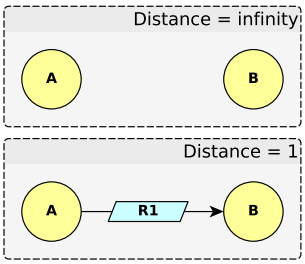

Figure 31: Distance between nodes

$$Dist(Favorite, B) = 1 \quad (5)$$

$$Dist(Label, A) = 1 \quad (6)$$

The interesting path however is $A \rightarrow Folder \leftarrow B$, which gives:

$$Dist(A, B) = 2 \quad (7)$$

Geolocation

Considering View 2 of Fig. 10:

$$Dist(Aircraft, PositionAircraft) = 1 \quad (8)$$

$$Dist(Pilot, PositionPilot) = 1 \quad (9)$$

This gives, considering $Aircraft \rightarrow PositionAircraft \leftrightarrow PositionPilot \leftarrow Pilot$:

$$Dist(Aircraft, PositionPilot) = 3 \quad (10)$$

Conclusion

The notion of distance can be a quite useful indicator. Create a path is often equivalent to go from $Dist(A, B) = \infty$ from $Dist(A, B) = 1$, 2 or 3 (see Fig. 31).

This means that, in a graph-oriented programming enterprise software, every node is a "virtual neighbor" of every other node: any change to the data can connect two nodes.

Distance can also be used to measure the optimal workflows in the user interface. If the developed process to go from A to B has a distance of 5, and most user create a shortcut between step 2 and step 6, to reduce the distance to 2, then we can conclude that the software missed some important functional point.

10.2 Adherence Between Domains

The connectivity will be at the center of some questions if we want to separate interconnected domains from a single database. The number of connections between domains could be a good indicator. We can define Adh_M for the measure of domain adherence in the graph model and Adh_D for the calculation of domain adherence inside the data.

We can take a sample of use of Adh_M. The Fig. 22 shows:

$$Adh_M(Business, Utility) = 1 \quad (11)$$

For sure, $Adh_D(Business, Utility)$ will be much larger.

This indicator can also be used to characterize the adherence level of business domains. For instance, in Fig. 29, views 1 and 2:

$$Adh_M(Maintenance, Inventory) = 1 \quad (12)$$

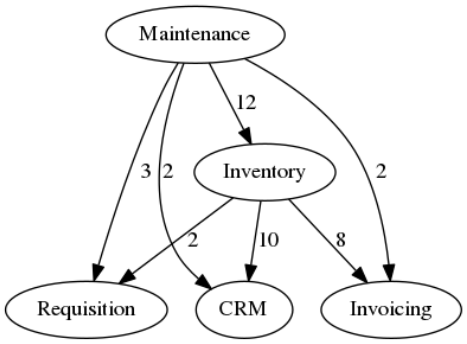

Figure 32: Functional map based on Adh_M

In view 3, we have:

$$Adh_M(Maintenance, Inventory) = 0 \quad (13)$$

The property (13) is very interesting because our design choice forwarded the adherence problem on the `Requisition` domain but separated semantically the domains `Maintenance` from `Inventory`.

This indicator can be generated from the graph-oriented modeler and will be helpful to determine if our semantic domains are correctly delimited. If we have in the model a very high value for $Adh_M(A, B)$, it is probable that A and B are indeed the same domain that the designer artificially split[63].

We can note that we can build a graph with domains represented as vertices and adherence represented as weighted edges (see Fig. 32). This can represent a functional and dependency macroscopic map of the software[64].

10.3 Graph Algorithms

Graph query languages [20] propose various ways of performing pattern matching inside a graph database. For enterprise software, pattern matching is not required in most use cases because we already know what we are talking about. Very commonly, we have a starting root note coming from a utility (user history, container, etc.) or the result of a search query from a business key.

Generally speaking, algorithms are not the priority of enterprise software (recommendation algorithms, shortest path, etc.). They are more used in Big Data databases for BI purposes.

List of Figures

1 Case of a simple aggregation `0..1` 2
2 Case of the $0..n$ aggregation 3
3 Evolution of the scope of an application 4
4 Evolution management in enterprise software . 6
5 Node and relationship types 9

[63] We have the same kind of measurement in IT architecture with Archimate modeling language. Refer to [17].

[64] What we obtain should be quite near from the functional view we can get with Archimate.

6 Node and relationship types with attributes . . . 9
7 Comparison between UML class diagram and graph-oriented modeling structural view 9
8 Representing aggregation in graph-oriented modeling 10
9 Representing specialization in graph-oriented modeling 11
10 Typed Attributes as Node Types 12
11 Domain concept illustration and inter-domain relationships 13
12 Instances 14
13 Instances linked to types 14
14 The generated model view centered on a node . 16
15 Representation of graph transformation 19
16 Subgraphs as containers 20
17 Rewiring preserves unknown relationships . . . 22
18 Field evolution, design 1 23
19 Field evolution, design 2 24
20 Field evolution, design 3 24
21 Strong typing of business node types and relationships 25
22 Decoration Utility: Attachment 26
23 Basic View Concept 27
24 Shortcut Utility 27
25 Container Utilities: Folders, Favorites, Labels . 27
26 The History Facility 28
27 Gluing Domains Together 28
28 Gluing with Supertype 29
29 Three Solutions for Gluing Business Domains . 29
30 GUI Opportunities For Alternate Navigations . 31
31 Distance between nodes 35
32 Functional map based on Adh_M 35

List of Tables

1 Comparison between database types 8
2 Sample of relationship constraints table 11
3 Minimum set of requirements for a graph-oriented implementation 16
4 Minimum set of requirements for a graph manipulation API 17
5 Pseudo code for the h graph transformation . . 22